Approved for public release; distribution is unlimited

RE-TARGETING THE GRAZE PERFORMANCE DEBUGGING TOOL FOR JAVA THREADS AND ANALYZING THE RE-TARGETING TO AUTOMATICALLY PARALLELIZED (FORTRAN) CODE

Pedro T.H. Tsai
Naval Research Lab, Monterey CA

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
March 2000

Author: _____
Pedro T.H. Tsai

Approved by: _____
Debra Hensgen, Thesis Advisor

_____
Rudy Darken, Second Reader

_____
Dan Boger, Chairman
Department of Computer Science

ABSTRACT

This research focuses on the design of a language-independent concept, Glimpse, for performance debugging of multi-threaded programs. This research extends previous work on Graze, a tool designed and implemented for performance debugging of C++ programs. Not only is Glimpse easily portable among different programming languages, (i) it is useful in many different paradigms ranging from few long-lived threads to many short-lived threads; and (ii) it generalizes the concept of intervals over Graze's original definition. Glimpse's portability has been validated by demonstrating its usefulness in performance debugging of both Java programs as well as automatically parallelized FORTRAN programs.

TABLE OF CONTENTS

LIST OF FIGURES

ACKNOWLEDGMENTS

I like to thank Debra Hensgen, this thesis would not have been possible without her guidance and teaching. To my parents, for their support and understanding. Finally, to Paul, who was my brother, mentor, and my best friend, with all my love.

I. INTRODUCTION

This thesis focuses on the design and implementation of a language independent performance debugging suite called Glimpse. Glimpse collects profiling data from the execution of multi-threaded programs and provides visualization tools to help programmers analyze the collected data. In particular, it builds upon the approach taken by Graze [Ref. 1], a performance debugging tool that monitors C++ programs that use the Solaris thread library. Glimpse generalizes Graze, making it both language independent and more functional. A language independent tool is needed to permit monitoring of many threaded programs important to the Navy as well as to all of DoD. Examples of such programs are the Master Environmental Library **(MEL)**, which is written in Java; and the US Navy Numerical Weather Prediction **(NWP)** applications, which are written in *FORTRAN*. **MEL** is a new system being developed by Naval Research Laboratory (NRL) for the purpose of disseminating environmental data over the network. It is important in such a system to dynamically identify performance bottlenecks. NWP applications, on the other hand, are legacy *FORTRAN* code that is being ported to high-end workstations (SGI, DEC/Alpha) and automatically parallized. Here bottlenecks may arise simply from executing the code on a new platform and, to make the job more difficult, the people porting the code are not, in many cases, the original programmers.

1. Motivation

The NWP applications, for example, the Navy Operational Global Atmospheric Prediction System **(NOGAPS)** [Ref. 2] and the Coupled Ocean Atmosphere Mesoscale Prediction System **(COAMPS)** [Ref. 3], predict atmospheric parameters such as winds, temperature, pressure, and precipitation. These predictions are based on the solution of sets of thermodynamic and fluid dynamic equations. By integrating these equations forward in time, NOGAPS and COAMPS are able to predict the state of the atmosphere in the near future. Because NOGAPS and COAMPS are compu-

tationally intensive, and the information they produce is perishable, minimizing the execution time of these codes is paramount. Most NWP applications use some form of parallel computing to improve their performance. Rather than reimplementing these applications in a parallel programming language, programmers often only add compiler directives to legacy code. These directives cause the compiler to automatically parallelize the code. Programmers place these compiler directives in what appears to be the comment sections of the source code, usually near the computationally intensive loop constructs. These directives indicate to the compiler that a particular loop may be safely parallelized. The compiler will then perform data flow analysis and attempt to generate the necessary machine code for the parallel execution of the loop.

MEL [Ref. 4], another example of an application that can benefit from performance monitoring, is an online digital library for environmental data and other resources. Environmental resources are often (i) difficult to locate, (ii) frequently duplicative, (iii) independently defined and formatted, and (iv) accessible only through an interface that is unique to each repository. The objective of MEL is to provide the user with a single interface that they can use to discover, query, retrieve, and order environmental data. Conceptually, the MEL system is analogous to the card catalog in a library. The card catalog enables the user to search the entire holdings of the library by specific criteria such as topics, authors and titles. Using the emerging standards for describing geo-spatial data and contents, MEL provides a digital metadata database for environmental resources. Since it is not practical to replicate the existing worldwide system of distributed repositories by creating a single, massive resource site for environmental information, the designers of MEL instead choose a three-tier client-server architecture. The first tier is the user interface to the MEL system. It consists of customers and standard WEB browsers. The second tier is the MEL access site, which consists of both hardware and software. The MEL access site fulfills the dual role of handling incoming queries and orders from the customers,

as well as matching the queries and orders to MEL software resident on the third tier. The third tier is composed of actual resource site databases, including their extraction and delivery processes. MEL software is installed at each resource site to facilitate interactions between the MEL access site and the resource sites. The MEL Service Architecture (**MSA**) is the software library (API's) developed to implement the multi-threaded servers running on each MEL access site and MEL clients running inside the WEB browser. MSA uses the Common Object Request Broker Architecture (CORBA) framework and communicates over the network via the Internet Inter ORB Protocol (IIOP). Because there are typically many users and only a few MEL access sites, the access site server that handles the user query and order is potentially a bottleneck. A performance bottleneck could also occur between the MEL access site and the resource site servers that process the requests.

MEL is an object-oriented client-server application, where as NOGAPS and COAMPS are scientific *FORTRAN* codes requiring substantial floating-point computation. Although the programming language used to implement MEL is quite different from NOGAPS and COAMPS, they share a common feature in that they both use a threaded programming paradigm. Unlike a sequential program, performance bugs in a multi-threaded program can be difficult to find using the conventional performance analysis tools. Particularly difficult to determine are performance bugs due to thread synchronization and communication. For example, in the automatically parallelized *FORTRAN* code, a computational loop might be distributed across several threads, with each thread working on a portion of the loop. Typically, there is an implicit '*barrier synchronization*' after the end of the parallel loop; a thread that completes its portion of work will wait until all other threads have finished their portion of work. If there is a load inbalance between threads, it is possible that some threads will idle for a long time while waiting for threads that are still working. Another type of performance bug is due to the communication overhead between threads. An idle thread might periodically query other threads to see whether they have completed.

Such activity, if performed frequently, adds significant overhead to the execution of parallel programs.

2. Background

Conventional performance debugging tools (such as *prof* and *pixie*) are designed to collect profiling information on sequential code. *Pixie*, an object instrumentation tool, adds profiling code to the executable. The modified executable is then run to generate information on code execution frequency. On the Silicon Graphics systems, *Speedshop* software (an integrated front end to *pixie* and *prof*) reads an executable, partitions it into basic blocks, and writes out an equivalent executable program containing additional code that counts the execution of each basic block. (A basic block is a region of the program that can be entered only at the beginning and exited only at the end). Data collected by *pixie* is then analyzed by *prof* that generates reports on the various statistics such as the frequency of function calls and total percentage of time spent in each function. These conventional tools can provide very useful and detailed information on how a program is spending its time, but they do not provide information on interaction between threads. For example, *pixie* would be able to tell us how much total time a program (all threads) spends in a parallel loop, but it cannot tell us whether some threads are spending too much time idling due to a synchronization barrier.

Parallel debugging tools such as Graze [Ref. 1], Pablo [Ref. 5], PSpec [Ref. 6] and Paradyn [Ref. 7] have attempted to solve these problems. The first three tools are user-controlled while the last, *Paradyn*, searches for types of performance problem which have, in the past, commonly plagued parallel applications. The user-controlled tools allow the user to collect profiling information during the execution of threaded programs and to visualize the information collected so that the user gains insight into the significance of data that was collected in this manner. Furthermore, these tools are more flexible than *pixie* in that they allow users to define which performance data is to be collected. In particular, *Graze*, *PSpec* and *Pablo* allow the user to define

events. An *event* has a name and at least one numeric attribute, the time at which the event occurred. When an event occurs, these systems record substate information corresponding to the event. An intrinsic attribute of an event is the identifier of the thread in which it occurred. This attribute allows the user to distinguish events occurring in different threads. Additionally, in *Graze* and *PSpec*, a user can define an *interval* by specifying a designated start event and an end event. The interval is the concept that allows the user to associate related events. In *Graze*, this association can be further specified by the user who requires that the two associated events have identical attribute values (with the exception, of course, of the thread identifier and time). By allowing the user flexibility in deciding what performance data are to be collected and visualized, experiments focusing on different events and intervals can be conducted.

While these tools are headed in the correct direction, more is needed. In particular, language independent techniques that are also independent of thread paradigm would be useful. Additionally, the interval concept needs to be generalized. Building on the approach taken by *Graze*, this thesis shows that the event and interval concepts can also be applied to both automatically parallelized *FORTRAN* program and to multi-threaded Java programs. A working prototype of such a generalization, *Glimpse*, that resulted from this research, is described.

3. Organization

The rest of this thesis is organized as follows. In Chapter II the background and related work on parallel performance debugging are described. Chapter III focuses on the approach used by *Graze*. Chapter IV discusses the changes needed to generalize *Graze* into *Glimpse*. In Chapter V, the results from testing *Glimpse* with a multi-threaded Java program are presented. In Chapter VI, we describe how the same generalization technique can be applied to automatically parallelized FORTRAN applications. That chapter also explains how *Glimpse* was used to gather and analyze performance data from a computationally intense NWP application, COAMPS. The

final chapter enumerates lessons learned while designing and implementing *Glimpse*, summarizes the contributions of this thesis, and describes suggested future work.

II. RELATED WORKS

In this chapter we describe the approaches others have taken in performance debugging of parallel programs. In particular, we discuss several software tools that are available in this application domain.

In section one, the traditional approaches for measuring performance of programs are discussed. In section two and three, we review two research software tools, *Paradyn* and *Pablo*, that represent more recent developments in the area of parallel program debugging. These tools implement dynamic instrumentation and automatic search for performance bottlenecks, intelligent data reduction, and the idea of adaptive control (self-steering) systems applied to performance optimization. In section four, we describe an integrated performance prediction, measurement, and analysis environment. In section five, commercial software, *TimeScan*, for debugging parallel code is described. We summarize these other approaches in section six.

1. Traditional Approach to Performance Debugging

Performance debugging can be characterized by the following activities: measuring performance data from the application, analyzing the data collected to identify bottlenecks, and optimizing the program code that causes the bottlenecks. In this section, we discuss several approaches that are used for data measurement and analysis, and the trade-off between each approach.

Program Counter Sampling. This approach is also known as *profiling.* It is widely used by the UNIX debugging tools such as `prof` and `gprof`. In this approach, a running process is periodically interrupted by the kernel (or some external process) to record a histogram of the program counter locations. Since each histogram bin can be related to a basic block (function) in the program, an estimate of the total time spent in a particular basic block (function) can be obtained by multiplying the corresponding histogram bin value with the known sampling period. Additional information such as call stack can also be recorded at the sampling point to provide

information about how the program counter gets there. This information allows the post-processing program to compute a duration known as the inclusive time of a **function**, that is the time spent in a function and all other functions that it calls. Without the call stack information, only the exclusion time is known. The sampling rate can be set by the user to control the resolution and the amount of data recorded.

On some systems, instead of using a timer, a hardware performance counter can be used to trigger the program counter sampling. For example, on the SGI R10000 systems, one can request the kernel to examine the program counter when the hardware performance counter specified by the user overflows, and record a histogram of the value of the program counter at overflow. The types of hardware performance counters that can be specified for the sampling purpose are counters that record TLB misses, primary or secondary data cache misses, primary or secondary instruction cache misses, etc.

The data collected by program counter sampling is statistical in nature, varying from run to run. Because the data collection process is external to the program being analyzed, no modification to the source code or the object code of the program is needed. This method has the advantage of low overhead when compared to other data collection methods.

Basic Block Counting. This technique counts the number of times that a basic block (function) in a program is executed. Because the counting is not a statistical measure, the observed frequencies are exact. However, the program needs to be instrumented with code to count the number of times each basic block (function) is executed. The instrumentation is typically performed on the object file. For example, on most UNIX system, an object instrumentation tool such as `pixie` reads the executable file, and writes out an equivalent file containing additional code that counts the number of times each basic block (function) executed. To obtain timing information, this technique must either periodically timestamp the recorded count data, or else use a machine model to compute the instruction cycles executed for each basic

block (function), and then infer the time spent in each basic block (function) from the number of instructions executed. The time estimate obtained using the machine model assumes that instructions are executed in an idealized condition.

Event Tracing. In this approach, a program is modified to include data logging code to record specific events during the execution. To instrument the program for event tracing, one can annotate the source code and then process the modified code through a pre-compiler, which translates the programmer's annotation into actual code. Another approach is to provide a set of event-logging API's and a library. The programmer can then instrument the application by inserting data logging calls at the appropriate locations in the code, and then compile the application code linking with the event-logging library.

The types of events that can be recorded include procedure entry or exit points, read and write function calls, or any other location specified by the programmer. Comparing to the profiling and basic block counting approaches, event tracing generates a complete sequence of events that describes the behavior of the program; thus it is the most general instrumentation approach. Because each event must be timestamped and recorded separately, and additional substate information must be recorded at each occurrence of an event, the potential data volume for the event tracing is large. Statistical techniques have been proposed as a means to reduce the amount of the data recorded while still provides an accurate description of the program behavior [Ref. 8].

2. Paradyn Parallel Performance Tools

Paradyn is a tool developed at University of Wisconsin-Madison for debugging parallel programs [Ref. 7]. *Paradyn* has the following characteristics: (1) it is designed to monitor long running program and large program with thousands of procedures; (2) it uses well-defined data abstractions to describe performance related problems; (3) it provides the ability to automatically search for bottlenecks that are known to affect the performance of parallel program; (4) it uses dynamic instrumen-

tation to instrument only those parts of programs relevant to finding the current performance problem; and (5) to leverage off existing visualization tools, *Paradyne* provides a standard interface to the performance data that allows the user to incorporate external visualization programs for examining the performance data. In the following discussion, we describe the approaches and components used by *Paradyn* to implement the functionality listed above. The components that make up *Paradyn* are the Performance Consultant, the Data Manager, the Metric Manager, the Instrumentation Manager, and the Visualization Manager.

Performance Data Abstractions. *Paradyn* uses two basic data abstractions for collecting, communicating, analyzing, and presenting performance data [Ref. 7]. The abstractions are the metric- focus grid and the time-histogram. A metric-focus grid consists of two orthogonal lists of information. The first list is a vector of performance metrics such as CPU utilization, memory usage, and counts of floating point operations. The second list, focus, is a specification of a part of a program expressed in terms of program resources. Typical resource types are synchronization objects, source code objects (procedures), threads, and processes. The combination of a list of performance metrics with a list of program resources forms a matrix (called a grid in *Paradyn*) with each metric listed for each program resource. The elements of the matrix can be single values, such as an average, a minimum or maximum value of a metric, or time-histograms.

A time-histogram is an array whose buckets store values of a metric for successive time intervals; *Paradyn* uses time-histograms to store metric values as they vary over time. The user can control the amount of data recorded and the resolution of the data by setting the total number of the buckets and the width of the bucket (i.e., the time interval). If a program runs longer than the bucket width times the number of buckets, *Paradyn* doubles the bucket width and re-bins the previous values. The process of doubling the width of bucket is repeated each time all of the buckets are filled. This re-sampling technique reduces the rate of data collection and allows *Para-*

dyn to monitor long-running programs while maintaining a reasonable representation of a metric's time-varying behavior [Ref. 7].

Automatic search of performance problems. To assist the user in locating performance problems in the program, *Paradyn* uses a well-defined notion, called the W3 Search Model, that organizes information about the types of problems found in programs and the various components contained in the current programs. Performance problems are found by searching through the space defined by the W3 model.

The W3 Search Model abstracts those aspects of a parallel program that can affect the performance into three domains: (1) *Why* the application performance is poor, (2) *Where* the performance problem is, and (3) *When* the problem occur. The "why" axis contains common types of performance problem that occur in parallel programs. These potential performance problems are represented as a set of hypotheses and tests. Each hypothesis can have sub-hypotheses, which narrows down the performance problem to a more specific aspect of the program behavior. For example, one hypothesis might be that a program is spending too much time on synchronization. The synchronization bound problem can be further attributed to two sub-hypotheses: (1) too many synchronization operations, or (2) high synchronization blocking time. By organizing classes of performance problems into a hierarchical order, the W3 search model allows the user to "drill down" to a specific cause of the performance bottleneck.

The "where" axis represents program resources in which performance problems lie. Searching along the where axis pinpoints the problem to a specific program component. Using the previous example, a "why" search might identify that a program is synchronization bound, a subsequent "where" search may isolate one synchronization object from among the many synchronization objects as the primary culprit. In *Paradyn*, the program resources are organized into different type of hierarchies, each resource hierarchy representing a related group of "focuses" that can be measured. For example, to identify which synchronization object is the primary

bottleneck, the search along the "where" axis starts the root of the hierarchy *SyncObject*. The next level contains different types of synchronization objects such as the *Semaphore*, the *Lock*, and the *Barrier*. Below the Lock and Barrier abstraction levels are the individual locks and barriers used by the application. The children of Semaphore are individual semaphores used in the application. Another example of a resource hierarchy is the *Procedure* abstraction, under which lies objects such as main.c, read_socket.c, write_socket.c, etc. Other *Paradyn* resource hierarchies include *Machine* (which contains sub-objects such as CPU 1, CPU 2, etc), *IO*, *Memory*, and *Process*. By abstracting different program resources into a separate hierarchy, *Paradyn* allows the user to concentrate on one abstraction at a time when searching for performance problems.

The third axis of the W3 search model is the "when"-which is used to identify at what time the application runs poorly. Programs have distinct phases of execution and the "when" axis represents periods of time during which different types of performance problems can occur. For example, a program may consist of three phases of execution: initialization, computation, and output. Within a single phase of a program, the performance tends to be uniform. However, when a program enters a new phase, its behaviors might change significantly. As a result, decomposing a program's execution into phases provides a convenient way for programmers to understand the performance of their program. Searching along the "when" axis involves testing the hypotheses for a focus during different intervals of time of the application execution [Ref. 7].

Paradyn's Performance Consultant module can automatically discover performance problems by searching through the space defined by the W3 Search Model. Refinements are made across the "where," "when," and "why" axes without involving the user. The search is conducted by considering a list of possible refinements along each axis, then ordering this list using internally defined hints. The Performance Consultant selects one or more refinements from an ordered list. If the selected refinement

is not true, the next item from the ordered refinement list is then evaluated. Paradyn can conduct a fully automatic search or allow the user to make manual refinements to direct the search.

The Performance Consultant is also responsible for directing the data collection process. It makes requests to the Data Manager and receives performance data from the Data Manager. The data collection process is described next.

Dynamic Data Instrumentation. Paradyn uses dynamic instrumentation to instrument only those parts of the program relevant to finding the current performance problem. Dynamic instrumentation defers instrumentation of the program until it is in execution and then inserts, alters, and deletes instrumentation during program execution.

Requests for dynamic instrumentation are made by the Data Manager in terms of a metric-focus grid. The requests are translated into instructions for insertion into the program. The translation is done in two steps. First, the Metric Manager translates the metric-focus requests into machine independent abstractions. Next, the Instrumentation Manager converts the machine independent representation into machine instructions for inserting into the application.

The machine-independent abstractions are expressed using points, primitives, and predicates. *Points* are locations in the application's code where instrumentation can be inserted (currently, the points understood by Paradyn's data collection facility are procedure entry, procedure exit, and individual call statements.) *Primitives* are operations that change the value of a counter and timer, e.g., set counter, add to counter, subtract from counter, set timer, start timer, and stop timer. Counter and timer are the two types of instrumentation supported by *Parady's* Instrumentation Manager: counter counts the frequency of some event in the application, and timer measures the interval between events. *Predicates* are conditional statements that guard the execution of primitives. They consist of a Boolean expression and an action. The Boolean expression can be computed using counters, parameters to a

procedure, return values from a procedure, or numeric or relational operators.

Paradyn's Instrumentation Manager performs the translation of points, primitives, and predicates into machine-level instrumentation. When Paradyn is initially connected to an application process, the Instrumentation Manager identifies all potential instrumentation points by scanning the application binary's image. Procedure entry and exit, as well as call to procedure are detected and noted as points. After Paradyn is connected to the application, the Instrumentation Manager waits for the requests from the Metric Manager. The requests are then translated into machine code fragments, called *trampolines*, for insertion into the binary imagery of the application process.

Two types of trampolines, base trampolines and mini-trampolines, are used. A base trampoline is inserted as follows. The machine instruction at the instrumentation point is replaced with a branch to the base trampoline, and the replaced instruction is relocated to inside the base trampoline. The base trampoline contains calls to mini-trampolines. The calls to mini-trampolines can occur both before and after the relocated instruction. A mini-trampolines is code that evaluates a specific predicate or executes a single primitive [Ref. 7]. Paradyn's Instrumentation Manager is responsible for generating the appropriate machine instructions for the primitives and predicates requested by the Metric Manager, and then transfering these instructions to the application process via a variation of UNIX `ptrace` facility. [1]

If the Performance Consultant determines from the data collected that a hypothesis is no longer valid, the primitives and predicates associated with testing that hypothesis can then be removed from the application process by the Instrumentation Manager.

Open Interface to the Performance Data. Once the instrumentation has been

[1] ptrace, process trace, is a UNIX system call that allows the parent process to control the execution of a child process. The parent can examine and modify the "core image" of a child process in the stopped state, and then cause the child process to continue. The UNIX debugging tool **dbx** uses ptrace to implement a breakpoint in the user's program.

inserted into the application, the data is sent back to the Data Manager for processing by other Paradyn modules. Paradyn provides a library and remote procedure call interface to access the performance data in real-time. Visualization modules (*visi's*) are external processes that use this library and interface. When a visi requests performance data from Paradyn, that request is sent to the Data Manager. If the request data is already being collected, the Data Manager will send the current values to the visi, and provide continuous updates as additional data are collected. If the requested data is no being collected, the Data Manager will request that the Instrumentation Manager start collecting it. Paradyn currently provides visi's for time-histogram plots, bar charts, and tables. The visi interface and library also can provide performance data for other uses, such as evaluating performance predicates for application steering, or logging performance data for experiments [Ref. 7].

In addition to Paradyn's basic data type, counter and timer values, performance data from external sources can also be collected. For example, some systems provide hardware-based counters that collect statistics on page faults, data cache misses, instruction cache misses, and memory usage activity. Data from these external sources can be integrated into the Paradyn instrumentation, and subjected to the same predicate evaluation as other performance metrics. For example, if a system provides a counter for the cumulative number of page faults in a process, then Paradyn's Data Collection Facility can read this counter before and after a procedure executes to determine the approximate number of page faults occurring in that procedure.

3. Pablo

Pablo was created by a research group at the University of Illinois at Urbana-Champaign. In this section we describe some of the work done by the Pablo group in the area of performance analysis of parallel systems. In particular, we describe these research on closed loop adaptive performance systems, and in the area of intelligent

performance data reduction to minimize the overhead associated with performance instrumentation.

The motivation behind the development of adaptive performance monitoring and application-steering systems comes from the following observations [Ref. 9]:

1) The traditional performance debugging processes are characterized by the following activities: (a) Application code is instrumented automatically by object code modifying programs or by compilers, or manually by inserting calls to the instrumentation library. (b) After instrumentation, performance data are captured from running one or more program executions. (c) The performance data are visualized and analyzed by the programmer to identify the bottlenecks. (d) Finally, based on measurement and analysis, either the program code that causes the bottleneck is modified or the runtime system policies are adjusted to better match the program resource requests.

2) As parallel computing evolves from homogeneous parallel systems to distributed collections of heterogeneous systems, application tuning and optimization problems become more complex. The time-varying resources of computational environments further exacerbate these problems. Moreover, the performance of parallel application is sensitive to slight changes in the application code, and to continually evolving system software.

3) Although effective for application codes with repeatable behavior, the traditional *post-mortem* tuning model is ill-suited to the parallel application with time varying resource demands that executes in a distributed heterogeneous environment. Not only may the execution context not be repeatable across program executions, resource availability could change during execution.

To address the issues of heterogeneous and dynamic computing environments, the Pablo group has developed a close loop performance analysis and adaptive control system. This system, called the "*Autopilot*" ([Ref. 8], [Ref. 9]), contains the following components:

(1) *Decision procedures* that determine how and when the system should adjust resource allocation policies and system parameters. (2) *Distributed performance sensors* that collect performance data for decision procedures. (3) *Resource policy actuators* that implement changes to the system parameters and policies in response to decisions.

Fuzzy Logic Decision Procedures. The *Autopilot's* decision procedures accept data from distributed sensors as inputs and use actuators to implement the results of decision processes. There are several traditional techniques for implementing such a decision mechanism, including decisions table and trees. A decision table for resource management would typically contain one dimension for each of the key performance sensor values (e.g., file read request sizes and cache hit ratios). Each dimension is then partitioned into a number of operating range (i.e., small, medium, and large read requests), and a policy and its associated parameters would be associated with each table entry. During uses, policies are identified via table lookup using the current sensor values.

Constructing a decision table to optimize performance presumes knowledge of precise mapping between the resource optimization policies and the sensor parameters. Furthermore, as the number of sensor parameters increases, the storage space to fully discretize the sensor space and associate policies can grow rapidly. Consequently, the designers of Autopilot choose to use fuzzy logic to implement decision procedures. The fuzzy logic system allows manipulation of linguistically described concepts through use of common sense knowledge, e.g., file prefetching benefits small, sequential reads.

Performance Sensor. The Autopilot performance instrumentation is based on a set of distributed sensors that extracts information from the execution application, which may be physically distributed over the network. A sensor has a set of associated properties that are defined at the time it is created. These properties typically include sensor name, type, network IP address, and any user-defined attribute-value pairs.

Sensors can collect data in either asynchronous or synchronous mode. In the asynchronous mode, a separate monitoring thread records the values of the program variables of interest at intervals specified by the client or at the time when the sensor is created. In synchronous mode, the sensors are inserted in either the source code or the object code of the application.

To reduce the amount of data collected locally by the sensor, a sensor can apply a data transformation function to raw data before recording them. The *attached functions* can compute simple statistics (e.g., sliding window averages) or more complex transformations. For example, one type of transformation is to generate qualitative file access pattern descriptions from file input/output request measures (e.g., converting a sequence of file seek operations to sequential, strided, or random access description that characterize the file I/O performed by the application).

Two additional services, a naming service and a client service, are provided to facilitate communication between sensors and decision procedures, and between decision procedures and policy actuators. The naming service supports registration of remote sensors and actuators, and it handles property-based requests for sensors and actuators by the remote clients.

The client service of the *Autopilot* is based on the *Nexus* communication layer [Ref. 10]. Nexus creates a global address space that encompasses all processes executing on a network. Before a client can communicate with a sensor or actuator, it must first obtain the *startpoint* and *endpoint* of sensors or actuators. Similarly, sensors and actuators must obtain the *startpoint* and *endpoint* of their clients. The term *startpoint* and *endpoint* refer to an address in the Nexus global address space. Together these capabilities allow decision procedures to acquire and manage remote sensors and actuators without knowledge of their physical location or creation times.

Policy Actuator. Autopilot actuators allow clients to modify the value of application variables and to remotely invoke application-level functions. Typically, actuators are used to change the resource management policy (e.g, changing file caching

policies). Like the sensors, actuators have associated properties such as name, network IP address, and attached functions.

The following example (from [Ref. 11]) illustrates how the sensors, fuzzy logic decision procedures, and actuators might be used to control file prefetching in an adaptive input/output system. A fuzzy logic controller relies on fuzzy sets to represent the semantic properties of each input (sensor) and output (actuator). The input values of the fuzzy variables are then mapped to the output space by a set of IF-THEN rules.

Figure 1 shows the basic flow of information through the fuzzy logic decision mechanism. The *Autopilot* sensors provide a time-varying stream of file read access classifications. The fuzzification step converts sensor inputs to a value (HIGH, MEDIUM, LOW) for the `ReadClassification` fuzzy variable. The following set of simple fuzzy rules are used to determine the value of output fuzzy variable `PrefetchingFactor`:

```
if ReadClassification = SEQUENTIAL then
   PrefetchingFactor  =  HIGH
if ReadClassification = RANDOM     then
   PrefetchingFactor  =  LOW
if ReadClassification = UNKNOWN    then
   PrefetchingFactor  =  MEDIUM
```

After defuzzification, the value of the `PrefetchingFactor` defines the action taken by an *Autopilot* actuator to adjust the number of blocks that are prefetched.

The rule sets used by the decision procedures are architecture independent; neither the source of fuzzy inputs nor the sink of the fuzzy outputs is specified. The value of `ReadClassification` is an abstraction whose value can be bound to a sensor value, or a classification, or even the output of another decision procedure. Similarly, `PrefetchingFactor` is an abstraction of an actuator, with no implicit mapping. Furthermore, one can experiment with different sensors, choose different actuator policies,

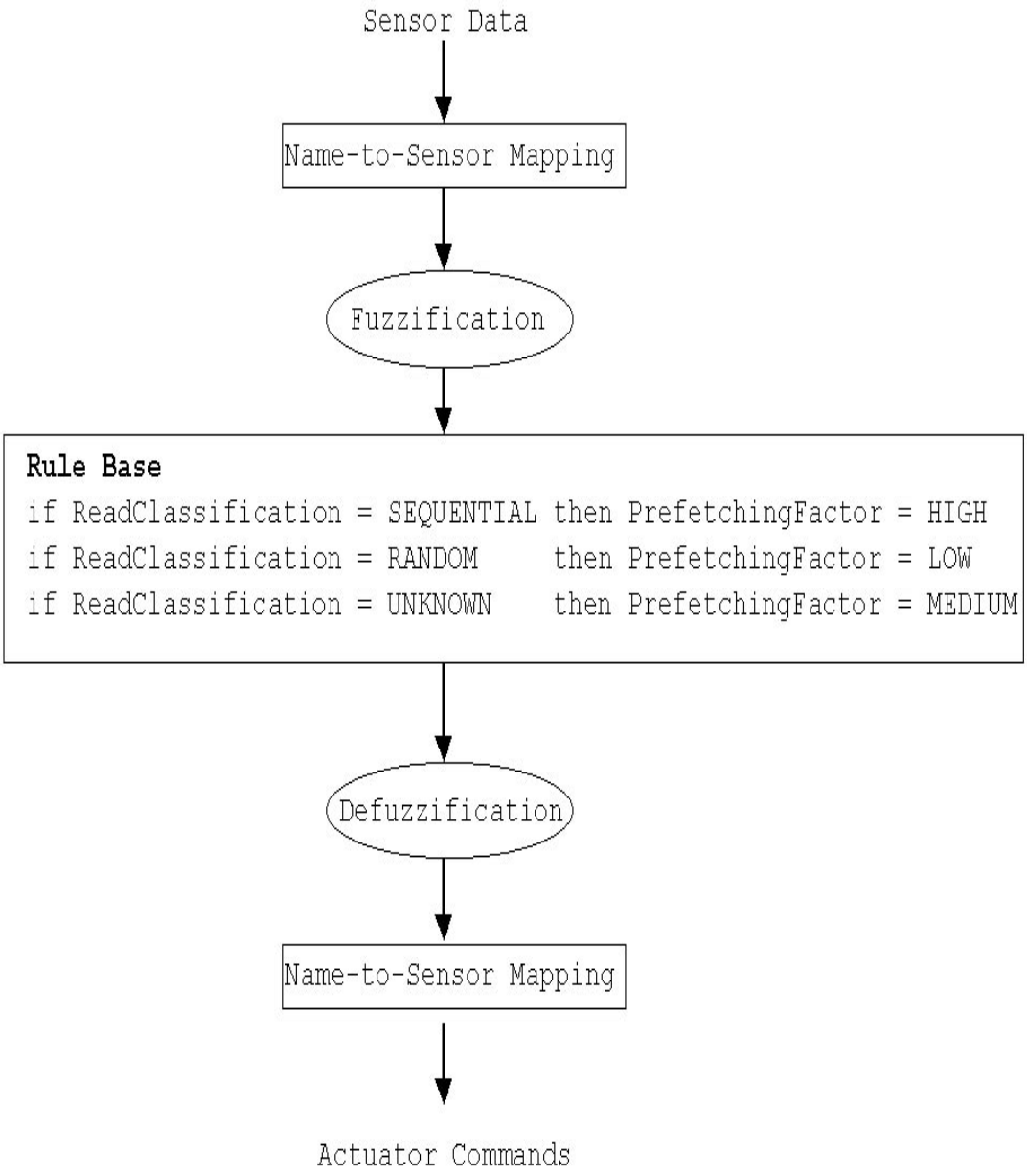


Figure 1. Data flow diagram for the Pablo's `Autopilot` decision mechanism.

or even control different systems by simply binding the inputs and outputs of decision procedures to different sensors and actuators.

4. Delphi

Delphi [Ref. 12] is a new performance environment under development by research groups from University of Illinois, Indiana University, University of Wisconsin, and the Argonne National Lab. Its aim is to provide an integrated performance prediction, measurement, and analysis environment for programmers to evaluate the software and hardware design choices, for both existing and proposed systems. These systems can range from a multi-threaded application running on single processor or multiprocessor machines, to object-oriented application running in a heterogeneous and distributed environment (e.g., CORBA applications). *Delphi* builds on the concepts, experience, and software from several current projects, including the *Pablo* and *Paradyn* performance analysis and measurement tools, the HPC++ and Polaris FORTRAN compiler systems, and the Globus metacomputing system.

An important concept of *Delphi* is the idea of performance prediction. To implement this capability, the *Delphi* framework includes: (1) compilers that emit code annotated with symbolic, execution-cost expressions, and embed calls to instrumentation library in the generated executable; and (2) computation models of key system components, including task schedules, memory, I/O component, network communication. *Delphi's* cost model can produce bounding estimates for the various phases of the program using the compiler- derived data on symbolic program variables and performance measurements from the execution of the instrumented code (the latter provide calibration data and an input-dependent aspect of program execution).

5. TimeScan

The *TimeScan* Event Analysis System is a commercial software tool for debugging, analyzing, and tuning the performance of single or multi-process programs, including programs using light-weight threads. The TimeScan software [Ref. 13] con-

sists of two components: (1) an event logging library (ELOG), and (2) a TimeScan viewer that displays event and state information. The following describes the basic concepts used in the TimeScane software:

event. Events are any program actions, value changes, or procedure calls that a user wants to monitor in order to understand the behavior of the program. For TimeScan, each type of event is identified by an event ID (a unique integer value). All events with the same event ID are interpreted as an *instance* of the same type of event. When events are recorded using ELOG library, ELOG stores a record of data for each event instance. Each logged event record contains a timestamp, an event ID, a data type tag, and a user data item.

event trace. An event trace contains event records from a process (or a thread) in the order in which they occurred. Each event trace is uniquely identified by the hostname, process ID, and thread ID (for program using light-weight threads.)

state. States represent time-spans that are marked by a starting and ending event. For example, by defining a state between a lock-request and lock-granted event, one can determine know how much time a program is waiting for a lock. Currently, only events occurring in the same event trace can be used to form a state.

event log. A log file contains one or more event traces and state information.

To use the TimeScan Event Analysis System, a user would perform the following steps [Ref. 13]:

1) Use the ELOG library functions [Ref. 14] to instrument the program, and compile and link the program with the ELOG library. 2) Run the instrumented program to generate the event log. 3) Run the TimeScan viewer to examine and analyze the event log and state information.

The ELOG library provides functions (for C and C++ programs) to initialize a log file, define event name and ID, define state, log event, and to handle various error conditions. The following pseudo C-code shows how one might instrument a multi-process program with ELOG:

```
#include <stdio.h>
#include ``elog.h''   /* Header file for ELOG function prototype. */

#define START_SEND  1      /* Define event ID */
#define END_SEND    2

int main(int argc, char* argv[])
{
/* Create Event Log. */
ELOG_INIT(``network.elog'');

/* Register Events */
ELOG_DEFINE(START_SEND,``Start writing to socket'',0);
ELOG_DEFINE(END_SEND,``End writing to socket'',0);

/* Define a State called SENDMSG between events START_SEND and
   END_SEND. */
ELOG_DEFINE_STATE(START_SEND, END_SEND,``SENDMSG'');

/* Create 4 child processes to do the work. */
for ( i = 0 ; i < 4; i++ )  {
    status=fork();
    if (status == 0 ) break; /* child process */
}
if (status != 0 ) exit(0);  /* Parent exits. */

/* Allocate a memory buffer for the event trace.  Each thread or
   process has its own memory buffer. */
ELOG_SETUP(``network.elog'',0,0);

/* Log the Starting event. */
ELOG_LOG(START_SEND, 0);

write_to_socket(data);

/* Log the ending event. */
ELOG_LOG(END_SEND, 0);

/* Flush the buffer to event log file. */
ELOG_OUTPUT();
}
```

The call to ELOG_INIT initializes the log file. Next, we defined the START_SEND and END_SEND event, and a state bounded by the two events. After the call to `fork()`, the parent process exits and the four child processes continue to execute. The first thing the child process does is to call ELOG_SETUP. The ELOG_SETUP allocates a memory buffer to store events for that given child process. The first argument to ELOG_SETUP specifies the log filename of the memory buffer to write to. (In this example, a single log file is used to store event trace from the four processes.) The second argument specifies the tag to associate with the memory buffer. A value of 0 causes the use of the default tag, which is derived internally by the ELOG library from the process and thread ID. The third argument specifies the size for this buffer. Using the default value of 0 will allocate space for approximately 1000 events. After the memory buffer is set up, calls to ELOG_LOG add the event instances to the buffer. Finally, ELOG_OUTPUT flushes the content of buffer to the log file.

Once the log file is generated, the user can visualize and analyze the data using the TimeScan viewer. The TimeScan viewer provides a facility to display event records and states as a function of time, with the Y-axis representing the trace from a different process or thread, and X-axis representing the time of the program execution. The viewer also provides support for viewing a subset of events and states, for editing the display symbol and color of the events and states, and for displaying histograms of state durations [Ref. 13].

6. Summary

In this chapter, we provide an overview of some of the existing tools for performance debugging and analysis of parallel programs. In particular, the dynamic instrumentation and automatic search of performance bottlenecks technique implemented by *Paradyn*, and the adaptive control and steering system for performance optimization by *Pablo* are discussed. Recent research efforts (*Delphi: An integrated, Language-Directed Performance Prediction, Measurement and Analysis Environment*,

see [Ref. 12]) aim to combine both dynamic instrumentation and automatica search for performance bottlenecks (W3 search model), with adaptive resource management, compiler integration, and performance prediction capability into an integrated environment.

THIS PAGE INTENTIONALLY LEFT BLANK

III. BACKGROUND ON GRAZE

Graze is a framework for the collection, visualization, and analysis of performance data from applications with multiple threads of control. It was developed at the University of Cincinnati and has been successfully used to identify performance bottlenecks in a multi-threaded VHDL simulation application [Ref. 1]. Unlike the traditional profiling tools such as *prof* and *pixie*, *Graze* lets the user specify exact locations in the program and types of run-time data to be recorded. The data collected by *Graze* are thread specific, that is data can be attributed to a specific thread of the monitored program. The ability to examine user-specified run-time data from the applications and to correlate information from different threads is important for identifying possible performance problems such as the overhead due to communication and synchronization among threads in a program. Such problems are difficult to detect from the conventional profiling data.

The *Graze* framework is comprised of three components: a specification language, a data collection facility, and a generic data visualization facility. In the following sections, we will describe each component of *Graze* in more detail.

1. Graze Specification Language

The *Graze* specification language [Ref. 1] is similar to the performance specification language used by PSpec. PSpec is a system designed for automated performance verification. PSpec uses assertion checking. Its specification language is designed for specifying performance assertions that are checked at run-time [Ref. 15]. For example, the following PSpec specification defines two events, an interval, and an assertion to check the performance of an I/O operation executed by a program:

```
timed event StartRead(); EndRead().
interval Read = s:StartRead, e: EndRead
         metrics time= ts(e) - ts(s)
         end Read.
assert { & r:Read: r.time < 10 ms }.
```

In this example [Ref. 15], `StartRead` and `EndRead` are declared to be timed events, which means they have implicit timestamp attributes. A `Read` interval starts with an event of type `StartRead` and ends with the next event of type `EndRead` after the start event. The variables `s` and `e` signify the start and end events for an interval of type `Read`. Each `Read` interval has a `time` metric whose value is the difference of its start and end timestamps (`ts` is a PSpec built-in function that returns the timestamp of an event.) The `assert` statement checks the elapsed time of any `Read` operation performed by the program. The statement `assert { & r:Read: r.time < 10 ms }` can be read as: "for all intervals, r of type `Read`, the value of r's `time` metric is at most ten milliseconds."

The *Graze* specification language uses several concepts from the PSpec language; in particular, the `event` and `interval` constructs. An `event` denotes a specific point of interest during the execution of an application; it has a type name and associated attributes. Every *Graze* event has two intrinsic attributes: its owner (a value that identifies the thread that produced the event) and a timestamp. Additional attributes can be specified by the user to cause *Graze* to obtain more detailed run-time information. Using a message passing application as an example, the following entries specify events corresponding to the preparing, sending and receiving of a message [Ref. 1]:

```
event Preparemsg() = diamond;
event Sendmsg(src, seq) = plus;
event Recvmsg(src, seq) = box;
```

The arguments `src` and `seq` are user-specified attributes that identify the sender and sequence number of the message sent and received. There are no hard-coded limits on the number of such attributes that an event type may have, however, in the original design of *Graze* these attribute values are restricted to the integer type. The tokens after the '=' are directives to the *Graze* visualization programs; in this case they specify the graphical symbol for depicting the respective event type.

A user can associate two related events by specifying an interval definition. An interval is bounded by a designated start and end event type. For example, the following specifies the interval between `Preparemsg` and `Sendmsg` events:

```
interval Write [ p: Preparemsg -> s:Sendmsg ] = line;
```

In the `Write` interval defined above, the syntax `p:Preparemsg` $\rightarrow$ `s:Sendmsg` tells *Graze* that after it finds each `Preparemsg` event it should find the next `Sendmsg` in the same thread. Each matching pair of `Preparemsg` and `Sendmsg` events make up an instance of interval `Write`.

In addition to the above interval specification, *Graze* provides another way of specifying the bounding condition between two events. Unlike the previous definition of the `Write` interval, this bounding condition does not require that the start and end events occur in the same thread. Such an interval specification is given below:

```
interval Transit [ s:Sendmsg -> r:Recvmsg ] Match = line;
```

The keyword `Match` tells *Graze* to create instances of interval `Transit` by matching all attribute values of the start and end events, except the timestamp and owner of that event (thread identifier). In this case, the bounding condition is equivalent to `s.src==r.src && s.seq==r.seq`. This type of interval specification is used for describing related events occurring in different threads. For example, in the message passing application, messages are typically transmitted by one thread and received by another thread within the same application. Therefore, using the `Match` interval specification is more appropriate.

Given an interval specification, *Graze* allows for either forward or backward searching of ending events. For example, in the following interval specification:

```
interval BackTransit [ r:Recvmsg <- s:Sendmsg] Match = line;
```

The left arrow symbol tells *Graze* to search backward in time from the point of a `Recvmsg` event instance until a corresponding `Sendmsg` event with matching attribute

values is found. If a matching event is found, then the *Graze* visualization programs marks this pair as an interval. These programs then move forward to the next starting event and repeat the backward search for a matching event, until all intervals meeting the specification are found.

As discussed in the next two sections, in addition to the specification language, *Graze* consists of a Data Collection Facility and a number of visualization toolkits. The specification language described in this section is used by both of these components, although some of the language is ignored by the Data Collection Facility. For example, interval specifications and event shapes are ignored by the Data Collection Facility because they are irrelevant during the run-time of the job being executed.

2. Data Collection Facility

Using the specification provided by the user, the *Graze* framework automatically generates a custom data collection facility for an application [Ref. 1]. A *Graze* utility program `builder` (written using lex and yacc tools) parses the specification and generates a logging function for each event type. The function name is simply the event type name with the suffix "_Stamp" appended to it. If the event type has optional attributes, then the function has formal parameters matching those attributes [Ref. 1]. For example, the C++ function prototypes for the `Sendmsg` and `Recvmsg` events discussed above are:

```
void Sendmsg_Stamp (int src, int seq);
void Recvmsg_Stamp (int src, int seq);
```

The actual code created by the `builder` program is system- and thread-library dependent. The original version of *Graze* only generates logging functions for multi-threaded program that use SUN Solaris thread library. The logging function writes event data to a log file. A logging function, when invoked, performs the following actions: (1) determines the thread from which it is being called; (2) finds the open file descriptor for the log file associated with the calling thread; (3) invokes a system-dependent timing function to obtain a timestamp; (4) writes the event type, timestamp and actual

parameter values to the output file. To minimize the overhead associated with the data logging, *Graze* implements an internal buffer for caching the event data. When the buffer is full, the data is written to the output file. This technique minimizes the number of writes performed.

To instrument an application, the programmer annotates the code using event notation. An example of event notation is:

```
Sendmsg(self, SequenceNumber);
```

A *Graze* specification language preprocessor should replace such lines with the call to the corresponding function, which, in this case is: [1]

```
void Sendmsg_Stamp(self, SequenceNumber);
```

After the application is executed, a set of log files is produced. Each log file contains information about events collected from a single thread in the instrumented program. The filename of the log file contains an integer that identifies the thread associated with the given event stream.

The structure of the log file consists of a header record, followed by a stream of event records. The integer value -1 is used as a sentinel to indicate the end of the log file. The header record consists of a single precision (4 byte) integer (0x12345670) and a double precision (8 byte) integer (0x123456789abcdef0). *Graze* visualization programs (`nibble` and `gorge`) read the header record to verify that input files are valid *Graze* log files and to determine the byte ordering (endianess) of the data. Each event record consists of an event type identifier, timestamp, and a list of integer values corresponding to the user specified event attributes, if any. All values in the log files are either single of double precision integers. The following layout shows the structure of the log file:

[1] Currently, the programmer must insert calls to the event logging functions, but it would be easy to modify the front-end of GNU compilers to perform this replacement.

```
header record:
             integer (4 byte)
             double precision integer (8 byte)

event records:
             event type identifier (integer: 4 byte)
             time stamp (double precision integer: 8 byte)
             optional program state information
             { integer, integer, ... }
             event type identifier (integer: 4 byte)
             time stamp (double precision integer: 8 byte)
             optional program state information
             { integer, integer, ... }
             .
             .
end marker:  -1
```

The event type identifier is simply a sequential integer value, starting at 0 that corresponds to the list of events given in the user specification. *Graze* uses this integer value to tag the event record in the log file rather than using the actual string name for both space efficiency and to minimize conversion problems caused by executing on one platform and displaying on another. The resolution of the timestamp depends on the operating system and hardware. On the Solaris system, a system call, **gethrtime()**, returns the current wall clock time. This value is expressed as nanoseconds since some arbitrary time in the past; it is not related to user CPU time or system CPU time as returned by other system calls such as **rusage()**.

3. Visualization Tools

Graze provides two tools for generic data visualization: `gorge` and `nibble`, that can be used once the performance data has been collected. *Gorge* displays the collected data with wall clock time increasing along the x-axis as shown in Figure 2. `Nibble` allows the user to graph generalized functions of statistical information pertaining to specific events and intervals as shown in Figure 3. Both visualization tools perform the following processing on the raw data: (1) they combine all of the data from the log files into a single event stream in memory; (2) they sort the event

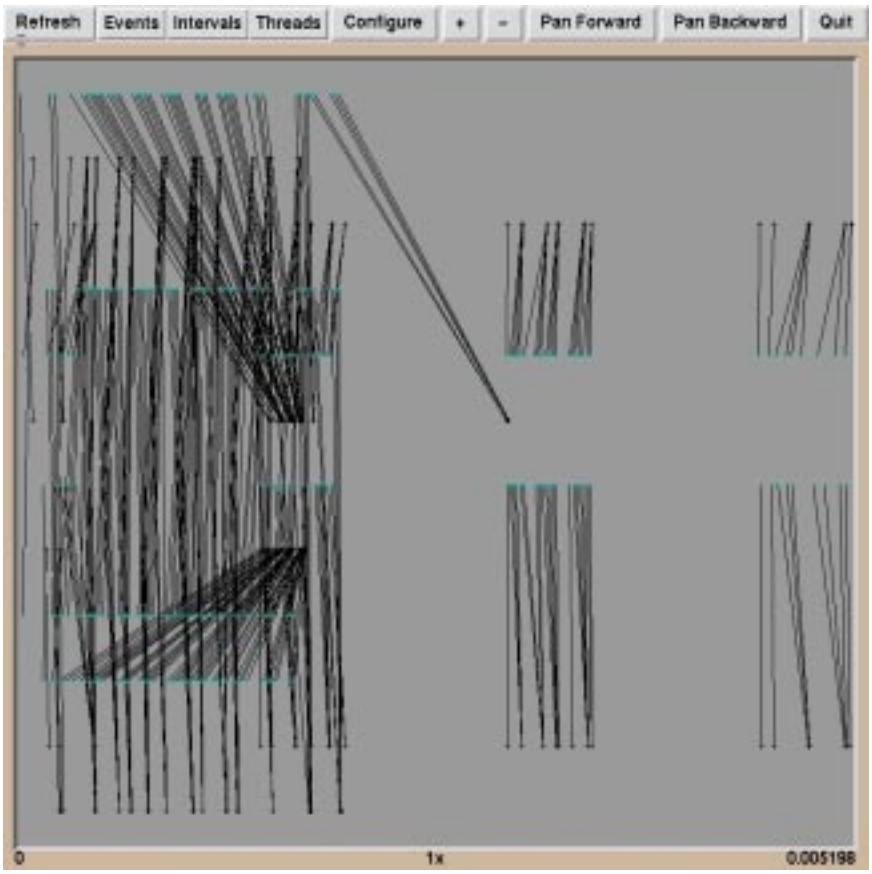


Figure 2. Gorge can be used to display the interaction between threads in a message passing application. For this example, event data from a total of 12 threads are shown, with event data from thread 0 plotted at the top of the graph and event data from thread 11 plotted at the bottom of the graph. The connecting lines between threads indicate the previously defined `Transit` interval. Time increases along the x-axis.

stream by time; and (3) they normalize the timestamp values so the first event in the event stream has the value of 0. Once the data have been preprocessed in this way, the visualization tools generate interval instances from the event stream by applying the matching conditions as defined in the user specifications.

As shown in Figure 2, the `gorge` tool displays data using a time-space graph, where the horizontal axis is the normalized time and the vertical axis represent data from each threads. The user can control the amount of information displayed by zooming in or out on the graph, and by selecting particular event, interval, and

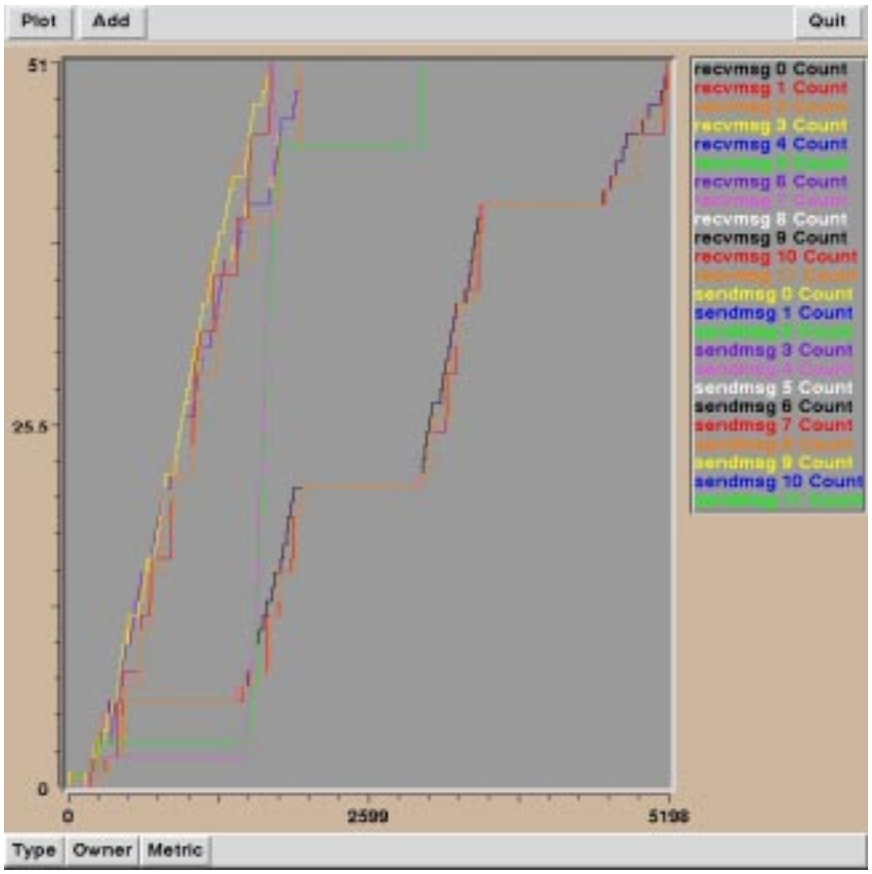


Figure 3. Nibble graphing the number of `Sendmsg` and `Recvmsg` events as a function of time in a message passing application. Time increases along the x-axis.

thread combinations [Ref. 1]. The `gorge` tool can provide a visual representation of the interaction between threads (Figure 2) by displaying intervals as a group of inter-connected lines.

As shown in Figure 3, the `nibble` tool is a generic statistical graphing tool for plotting data versus time. The types of data that can be graphed include quantities such as the total number of occurrences of an event or interval, a histogram of an event's state attributes, and the average elapsed time of an interval (Figure 3). The `nibble` tool provides mathematical operators that can be applied to the event or interval data to calculate new statistical data types on the fly for display. For example, graphs can be added, subtracted, multiplied, divided, and smoothed (Figure 4).

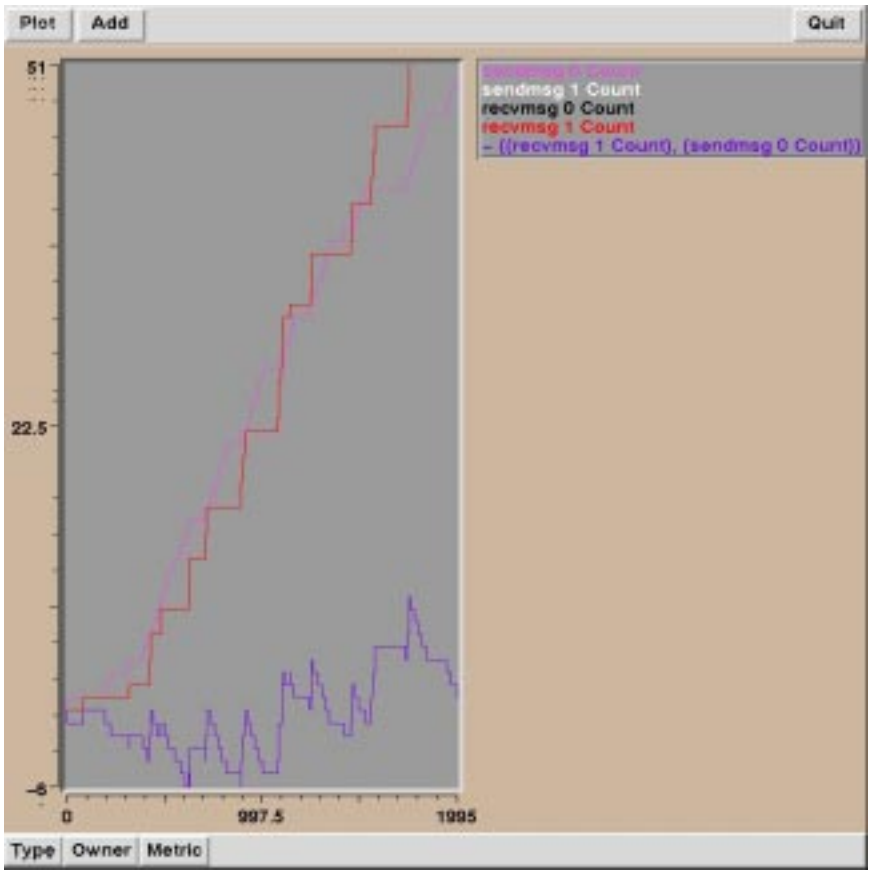


Figure 4. Nibble showing the number of `Sendmsg` and `Recvmsg` events in thread 0 and 1, and the differences (the lower line graph) between the number of messages received by thread 1 and number of messages sent from thread 0. Time increases along the x-axis.

4. Summary

In this chapter, we describe the three components of *Graze*: the specification language, the Data Collection Facility, and the visualization tools. The specification language lets the user defines `event` and `interval` types that are of interest for a given application. The specification defined by user is used to drive Data Collection Facility and visualization tools. In particular, the `event` specifications are automatically translated into data logging functions by the Data Collection Facility. The programmer then annotates the application with the `event` point by inserting data logging functions at the appropriate locations in the program. The data collected

from running the instrumented program, along with intervals constructed from the event data and the user-provided specification, can be plotted and analyzed by `nibble` and `gorge` tools.

IV. GLIMPSE: GENERALIZING GRAZE AND APPLYING IT TO MULTI-THREADED JAVA PROGRAMS

This chapter describes the design and implementation issues faced when generalizing *Graze's* Data Collection Facility. In particular, it addresses those issues faced when attempting to monitor multi-threaded Java programs (Appendix A).

The first section of this chapter provides an overview of the design issues encountered when generalizing *Graze* to apply to Java programs. These issues include mapping Java threads to event logs, accessing a high-resolution timer from within a Java program, and generalization of the *interval* definition. The next section describes event collection code generation, where *Graze's* events are translated into Java methods. The final section summarizes the design issues faced in implementing *Glimpse*.

1. Design Considerations

As described in the previous chapter, *Graze* provides a utility program that reads the user's specification and automatically generates C++ functions. The user then manually inserts events into the application. At execution time, event information is written to a log file. *Glimpse* provides similar capability for event information collection to Java programs. The following design goals and constraints were considered when implementing the *Glimpse* data collection facility:

i) We want to re-use *Graze* visualization tools; this implies maintaining the same data file structure and log filename convention in order to be compatible with the existing visualization programs.

ii) The overhead incurred due to the data collection code should be kept to a minimum. The modifications to the monitored programs should consist of a method call to initialize the data collection package, method calls to write out event data,

and a method call to free up resources used by the data collection package prior to program termination.

iii) To improve code modularity, the data logging methods, which depend on the user specification and are called (referenced) directly by the application program, should be placed in a separate module from the rest of the data collection package. This separation has the advantage that if the types of events to be monitored are changed, only the module containing data logging methods needs to be re-generated and re-compiled.

Based on these criteria, we implemented a new front-end program, **builder_java** and a new utility library, **glimpse.jar**. The **builder_java** program parses the specification and generates a Java class that contains data logging methods. The **glimpse.jar** library contains utility functions such as those needed for opening the log files, writing to the log files, and obtaining timestamp information. Several issues arose as we implemented the *Glimpse* data collection facility. The issues, and approaches we adopted, are described in the following sections.

a. Thread Naming

Graze stores event data from each thread in a thread-specific log file. To keep track of the log file that a data logging function is writing to, *Graze* uses the thread identifier provided by the system. The value of the thread identifier, which is of type integer, is obtained at runtime by calling a SUN Solaris thread library function. Based on this integer number, *Graze* determines the appropriate log file to which the event data is written. For every thread from which logging functions are called, there is a corresponding log file with the name of "log.{thread id}". This naming scheme is also shared by *Graze's* visualization tools, which expect data files to have names such as **log.2**, where **log.2** contains event data for thread 2, and where the log file for the first thread must be **log.0**.

This filename convention assumes that the system thread library uses a sequential numbering scheme, starting at 0, to keep track of threads that are created

by the program. It also assumes that threads used by the application exist for the duration of the program execution. By default, the Java thread library uses a string value such as "Thread-6" to represent a thread's unique identifier. The initial value of the integer in this string is system dependent. In some implementations of the JVM this number starts at 1 (Win95, Sun Java version 1.1.5), and in others it starts with 4 or 5 (Silicon Graphics IRIX 6.2, Java version 1.1.6). Furthermore, many threads in Java programs are short-lived. These are transient threads, created to perform a specific task in the application program. Once that task is completed, the thread exits the method it was created to execute and is garbage collected by the Java Virtual Machine. Threads created to perform asynchronous input and output operations are an example of short-lived threads.

Because threads in Java are inexpensive to create and destroy, an application program can potentially use a large number of transient threads. After a transient thread has exited, the Java thread library can re-use that same thread name again for a newly created thread. Directly translating *Graze* code to Java could therefore result in event data from two different threads being saved to the same log file. To resolve this thread names issue in a way that is consistent with *Graze's* current log filename convention so the existing visualization tools can read the data without modification, we designed and implemented a mechanism to associate each unique Java thread with its own log file.

One possible solution we considered was to require the monitored program to explicitly identify each thread that it creates with a unique integer, starting at 0. This approach will certainly prevent two threads from having the same name and thus saving event data to the same file. However it would mean calling the **setName(int id_value)** method (from the **Java Thread Library**) every time a thread is created in the application. This approach would likely add additional code to the monitored program as most programs do not explicit set the thread name but rather let it default. Additionally, this solution shifts the task of creating and managing the

unique thread name to the programmer. Therefore, this solution would make using the *Glimpse* Data Collection Facility unnecessarily tedious.

The approach we eventually adopted was to provide a mechanism for mapping the Java threads to unique log files that is transparent to the user (See Figure 5). The thread name to log file mapping is implemented using two utility classes: `ThreadData` and `ThreadPool`. The `ThreadPool` object maintains a pool of `ThreadData` objects. The `ThreadData` object contains various thread-specific information such as the actual name of the thread and the associated log filename. It also contains methods for initializing the log file, writing data, and closing the log file. With this solution, each time that an event logging method is invoked, that method sends a request to the `ThreadPool` object. The `ThreadPool` object returns a reference to the `ThreadData` object assigned to the thread that invoked the event logging method. If the `ThreadData` object for the calling thread is not found, then a new `ThreadData` object is created and added to the pool, and the reference to that newly created object is returned. The event logging method can then invoke the output methods of the `ThreadData` object to write out the corresponding timestamp and the user-defined event attributes to the appropriate log file. Any future write requests by the same thread will result in the `ThreadPool` returning a reference to the designated `ThreadData` object.

To ensure that every `ThreadData` object is assigned to a different log file, the `ThreadData` class maintains a global integer counter that is shared by all objects of this class. This counter is initially set to zero, and its value is incremented by one when a new `ThreadData` object is created. To ensure thread-safe behavior, the action of obtaining and incrementing this counter value by a thread is mutually exclusive from other threads that perform the same action. This mapping scheme isolates the thread name used by the Java program from the actual log filename, and preserves the original log filename convention.

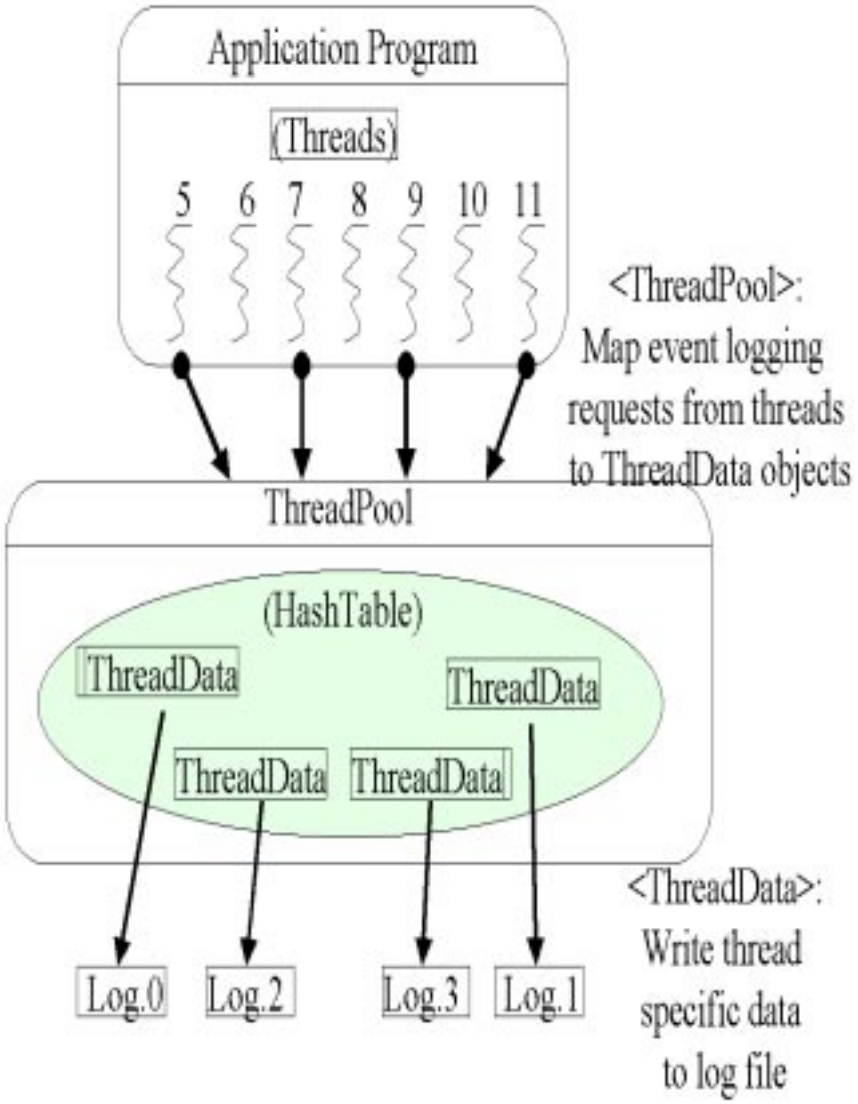


Figure 5. Glimpse's Java utility classes for mapping threads to log files.

To prevent saving event data from two different threads to the same file, as in the cases of short-lived threads with duplicate thread names, the only action required by the application is for it to notify the `ThreadPool` object when a transient thread is terminated. This notification is accomplished through the `closeThreadData()` method that is executed immediately before a thread terminates. This method removes the corresponding `ThreadData` object from the pool. Then, when another thread with the same name appears, the `ThreadPool` object will create a new `ThreadData` object and assign it a different log filename. [1]

b. High Resolution Timer

The Java core package provides a timing routine `currentTimeMillis()` in the `java.lang.System` class. This method returns the time, in milliseconds, between the current time and the standard base time known as "the epoch," 00:00:00 GMT on January 1, 1970. For performance monitoring with Graze, we need to record the real time at which events occur during the program execution. For some applications, millisecond resolution of the standard Java timing routine is too coarse. We investigated other alternatives for implementing a high resolution timer. One package we looked at is the PortableTimer from the **PTOOLS** working group [Ref. 16]. The PortableTimer specifies a set of timing functions that vendors should provide on their respective systems to facilitate performance measurement, and a sample implementation of the specification for several of UNIX platforms. The sample implementations use the standard UNIX functions to obtain wall clock, user and system time. Unfortunately, implementations of this specification are not yet available for our platforms.

Consequently we investigated building a custom timer. On the platform (Silicon Graphics R10000 architecture) where we did most of the development and

[1] We note that in the current implementation of Glimpse, that the user must explicitly call `closeThreadData()`. A commercial version of *Graze* could incorporate this call into the Garbage Collector. Another possible solution would be for the Java Thread API to provide a way to register a function that is invoked when a given thread exits. The `ThreadData` object would then automatically register a `closeThreadData()` method, and thus eliminates the need for the user to explicitly call this method.

code testing, a high resolution timing function is supported via a free-running 64-bits hardware counter. To use this hardware counter for performance timing, we wrote a custom timer that maps the location of the hardware counter to an address in the user process space. The value of the hardware counter can then be obtained by simply reading the value stored at that address. Depending on the particular version of the R10000 architecture, the hardware counter has a resolution (the elapsed time between ticks) between 21 and 800 nanoseconds. This resolution value can be dynamicallly determined by querying the hardware at run time.

To access this custom timer (which is implemented in the C language) from a Java program, we use the Java Native Interface (**JNI**) [Ref. 17]. JNI provides a standard mechanism for a Java program to access functions in machine binary code. The basic steps of using JNI are: (1) write a Java class that declares the C functions, with the appropriate return types and calling arguments and a keyword 'native' before the method name; (2) use the Java utility tool `javah`, to translate these Java methods to the equivalent C-language function prototypes and store them in a header file; (3) provide the implementation (in C or C++) of these functions as declared in the header file; and (4) create a dynamic shared library of these functions, i.e., the **.dll** library in the Windows NT environment or **.so** library in the UNIX environment. For example, the high-resolution timer used in our tests was implemented as follows:

```
final public class SystemTimer
{
..
/* Name of native function (implemented in the C-lang) that
   returns time in nanosecond.  The implementation of the
   function gethrtime_ns() must be provided in a dynamic
   shared library.  In this example this library is called
   libsgitimer.so . */
      public static native long gethrtime_ns();

/* Static Initializer: Load the dynamic shared library.  */
      static {
          System.loadLibrary("sgitimer");
```

```
      }
/* SystemTimer method that returns real time in nanosecond.
   It calls the native function gethrtime_ns().  */
      public static long gettime_ns()
      {
          return gethrtime_ns();
      }
..
}
```

The keyword `native` before the `gethrtime_ns()` method declaration tells the JVM that the actual implementation of this method in not in Java bytecodes, but machine binary codes. When the `SystemTimer` object is initialized, the JVM executes `System.loadLibrary("libname")` to load the dynamic shared library containing the high-resolution timer functions. The method `SystemTimer.gettime_ns()` simply calls the equivalent native function to obtain the real time in nanoseconds.

c. Generalization of the Interval Definition

We recall from the previous discussion of *Graze* in the last chapter that an *interval* is determined by two bounding events, and by the matching criteria, if specified, between the start and end events. The two criteria are to require that both events occured on the same thread, or to match all event attributes (except for the thread identifier and timestamps) of the two bounding events. For example:

```
event Sendmsg(src, seq) = plus;
event Recvmsg(src, seq) = box;
interval Transit1 [ s:Sendmsg -> r:Recvmsg ] = line;
interval Transit2 [ s:Sendmsg -> r:Recvmsg ] Match = line;
```

Interval `Transit1` requires that both `Sendmsg` and `Recvmsg` events occur in the same thread; interval `Transit2` requires that (`src` and `seq`) values are the same. We would like to generalize the *interval* definition so that it allows specification for partial matching of attribute values, and Boolean relationships between the event's attributes. This new syntax would support the following types of interval definitions:

```
interval Transit3 [ s:Sendmsg -> r:Recvmsg ]
         { s.src==r.src } = line;

interval Transit4 [ s:Sendmsg -> r:Recvmsg ]  = line;
         { s.src==r.src && s.seq >= r.seq } = line;

interval Transit5 [ s:Sendmsg -> r:Recvmsg ]  = line;
         { s.src!=r.src && s.seq < r.seq } = line;
```

Interval `Transit3` requires only the `src` variable of `Sendmsg` and `Recvmsg` to match. Interval `Transit4` requires the `src` value to match and that `seq` value of `Sendmsg` event is greater than or equal to the `seq` value of `Recvmsg` event. Under this expanded syntax (see Appendix B), the operators for comparing the attribute values are:

$$>, >=, ==, <=, <, != \tag{IV.1}$$

To support this more general form of interval specification, additional production rules are added to the lexical parser of the *glimpse*, and the internal data structure used by the semantic analyzer is expanded to include the additional clauses for evaluating the attribute data between the two events.

2. Event Collection Code Generation

We created a front-end program, **builder_java**, to automatically create application specific Java class that contains data logging methods. For each event type defined in the user specification, **builder_java** creates a corresponding static method. As an example we consider the specification for message passing events:

```
event Sendmsg(src, seq)
```

and

```
event Recvmsg(src, seq)
```

The Java class, and its methods, created from the above specification are:

```
final public class GzEvent
{
  static ThreadPool pool;
  final static boolean DISABLE_GLIMPSE=false;
  final static boolean NATIVE_TIMER=false;
  static SystemTimer st;

  static public void init()
  {
    if (DISABLE_GLIMPSE) return;
    pool=new ThreadPool();
    if (NATIVE_TIMER==true) st=new SystemTimer();
  }
  static public void init(String data_dir)
  {
    if (DISABLE_GLIMPSE) return;
    pool=new ThreadPool(data_dir);
    if (NATIVE_TIMER==true) st=new SystemTimer();
  }
  static public void close()
  {
    if (DISABLE_GLIMPSE) return;
    pool.closeAll();
  }
  static public long gettime()
  {
    long ts;
    if (NATIVE_TIMER)
        ts=st.gethrtime_ms();
    else
        ts=System.currentTimeMillis()*1000;
    return ts;
  }
  static public void Recvmsg(int src, int seq)
  {
    if (DISABLE_GLIMPSE) return;
    long ts=GzEvent.gettime();
    ThreadData handle= (ThreadData) pool.getThreadSpecific();
    handle.putout(0);
    handle.puttime(ts);
    handle.putout(src);
    handle.putout(seq);
```

```
    return;
  }
  static public void Sendmsg(int src, int seq)
  {
    if (DISABLE_GLIMPSE) return;
    long ts=GzEvent.gettime();
    ThreadData handle= (ThreadData) pool.getThreadSpecific();
    handle.putout(1);
    handle.puttime(ts);
    handle.putout(src);
    handle.putout(seq);
    return;
  }
  static public void closeThreadData()
  {
    if (DISABLE_GLIMPSE) return;
    pool.closeThreadData(Thread.currentThread());
  }
}
```

The name of the generated event logging class defaults to GzEvent, although the user can specify a different name by passing an optional argument to the **builder_Java** program. Calls to the `SystemTimer` methods to get timestamp values, and to write out the values of variables `src` and `seq` are automatically inserted into the body of the event logging methods, `GzEvent.Recvmsg(src,seq)` and `GzEvent.Sendmsg(src,seq)`. A flag, `NATIVE_TIMER`, can be set to allow the logging methods to use either the standard Java timing method, or the native timing function, if one is provided via the JNI mechanism as discussed in the previous section. To instrument the application, the programmer edits the source code and annotates it using `GzEvent.init()`, `GzEvent.Recvmsg()`, and `GzEvent.Sendmsg()`. [2] Two versions of the `GzEvent.init()` methods are provided–one takes an optional argument to specify the directory to which the log files are to be written. If not specified, the current working directory of the monitored program is used. As stated earlier, the

[2]In the current implementation the programmer must also manually insert a call to the `GzEvent.close()` method. See appendix C on using Java Finalize facility.

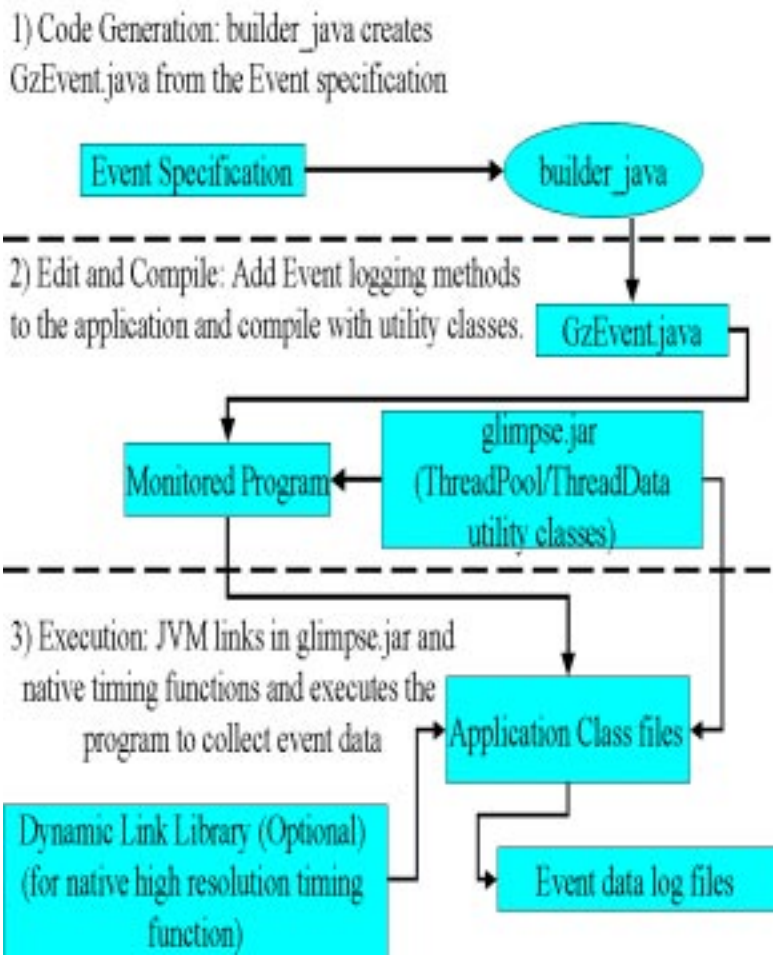


Figure 6. Steps for collecting event data: 1) code generation, 2) compiling the monitored program, 3) loading dynamic library during execution. Arrow indicates dependency at the various stages.

`GzEvent.closeThreadData()` method notifies the ThreadPool object that the calling thread is about to exit and that the log file associated with the exiting thead should be closed. This call is only needed to handle tranisent threads; it prevents data from two different threads being written to the same log file. Finally, `GzEvent.close()` terminates event logging and closes all log files. A flag, `DISABLE_GLIMPSE`, allows the user to turn off the data logging if the application no longer requires performance monitoring.

To compile the monitored program, both the `GzEvent.java` file and `glimpse.jar` utility library need to be supplied to the Java compiler (See Figure 6). The `glimpse.jar` library and dynamic shared library for the native high-resolution timer are then loaded in by the JVM during the execution of the monitored program.

3. Summary

In this chapter, we described the porting issues encountered while adapting *Graze* to monitor multi-threaded Java programs, and the approaches we used to resolve these issues. In particular, the issues with mapping application threads to event log filename, obtaining high-resolution timestamp information at run-time, and generalizing the *interval* definition. In the last section, we descibed how *Glimpse* translates user-defined event specification into the data logging code that is used to instrument the application of interest.

THIS PAGE INTENTIONALLY LEFT BLANK

V. EXPERIENCES USING GLIMPSE WITH JAVA

In this chapter we describe the result from testing *Glimpse* with a multi-threaded Java program. By visualizing the event data collected from execution of the test program and intervals derived from the event data, and by relating these events and intervals to the actions performed by a test program, we demonstrate that *Glimpse* is working properly.

The first section of this chapter describes the structure of a Java example program and the expected interactions between the various threads in this application. The next section describes the event and interval specifications that were used to collect the runtime data, and to visualize that data. Section three describes the data gathered from running the test program with two different inputs. By relating the events and intervals to the expected behaviors of the application, we demonstrated the correctness of the test program and *Glimpse's* Data Collection Facility. In the final section, we summarize the results from testing *Glimpse* with a multi-threaded Java program.

1. Description of the StopLight Program

To evaluate the *Glimpse* software, we applied it to a Java program called *"StopLight"* that simulates traffic flow at an intersection. This program was originally written to show how to use Java threads in a concurrent program. It uses car, intersection, and timer objects to model the flows through the intersection. Each car object is assigned a direction of movement (North-South or East-West) and a departure time. The departure time determines when a car will arrive at the intersection. The timer object controls the traffic lights of the intersection object. Each timer object is initialized with a timeslice value. When that timeslice value expires, the timer object toggles the traffic lights so that the flow direction at the intersection alternates between North-South and East-West. Two pair of timers are used to

control the traffic light at the intersection; their function will be described in next paragraph. Cars traveling in different directions can arrive randomly at the intersection, and more than one car can queue up at the intersection while waiting for the light to change. In this program, a car is not allowed to change direction once it starts running; furthermore there is only one intersection through which all cars must travel.

An intersection should maximize the number of cars passing through, and minimize the time that a car is waiting at the intersection. For example, the green light should stay on longer for the direction with a heavy traffic flow, and should turn red quickly if no more cars are passing through. To implement this behavior in the StopLight program, a timer called LargeTimer (that is a timer with a longer expiration value) and another timer called SmallTimer are used concurrently to control the traffic light. Both timers are started when the intersection object is first initialized. Each time a car passes through the intersection without stopping, the currently active SmallerTimer is stopped and a new one is started. Stopping and restarting this SmallTimer has the effect of prolonging the green light for that given direction. Thus, if a group of cars, traveling along the same direction, arrives at the intersection at close intervals, then the traffic light for that direction will tend to stay green. However, if there is a long gap between cars travelling along the same direction arriving at the intersection, then the SmallTimer will most likely expire before the next car can reach the intersection. When the SmallTimer expires it forces the traffic lights to change, thus allowing the cars traveling along the other direction to proceed. This control logic ensures that cars waiting at the intersection will not wait longer than necessary (not greater than the expiration value of the SmallTimer) if the cross traffic is very sparse.

The LargeTimer ensures that both directions are given their fair share of green light. For example, consider the case where there is a caravan of cars traveling along the North-South direction, while only a few cars are traveling along the East-West

direction. If the gaps between the vehicles in the caravan are very short and there were no LargeTimer object, it is likely then that North-South bound cars could keep extending the traffic light in their favor, and the East-West bound cars would not be able to proceed until all the North-South bound cars were through. The LargeTimer prevents this unfair situation from occurring. Regardless of the number of times that a SmallTimer is restarted, the LargeTimer will eventually expire and force the traffic light to switch. This design guarantees that the maximum waiting time of any cars will not exceed some threshold value, regardless of the uneven traffic distribution between the two flow directions.

In the implementation of the StopLight program, each car and timer object executes as a separate Java thread. Timer threads are assigned higher priority than car threads. The higher priority associated with the timer thread allows it to interrupt other threads in order to change the traffic light at the intersection. The critical sections of this program are associated with changing the traffic light variable of the intersection object. To ensure the synchronized access to the traffic light variable, the intersection object acts as the monitor to the critical sections. (See Appendix A for a discussion on the Java monitor object and synchronization.)

2. Defining Events and Intervals For the StopLight Program

To evaluate the *Glimpse* package with the Stoplight program, we define the following events that are relevant to the car, timer and the intersection objects. For the car objects, we defined three types of events: `StartMotor`, `ArriveIntersection` and `LeaveIntersection`. These events correspond to 1) when the car is started, 2) when it reaches the intersection, and 3) when the car has crossed the intersection. In term of actual location in the `StopLight` program code, these events correspond to 1) when the car object enters its `run()` method, 2) when the car object invokes the `enter()` method of the intersection object, and 3) when the car object returns from the `enter()` method of the intersection object.

For the intersection object we defined the `Stop`, `Go`, and `NoWait` events. The `Stop` event occurs when a car reaching the intersection has to wait for the light to change. The `Go` event occurs when a car waiting at the intersection can proceed. The `NoWait` event occurs when the traffic light direction and car direction are the same, in that case the car just proceeds through the intersection without stopping. The following code shows the locations of event collection points in the `Intersection` object:

```
public class Intersection {
    public int traffic_light ;
    public LargeTimer MajorTimer;
    public SmallTimer MinorTimer;
    . . .
    public synchronized void enter(Car cx, int car_direction)
    {
        /* car going at NORTH or SOUTH direction and so is the current
           traffic light for the intersection. */
        if ( traffic_light == NSDirection &&
             (car_direction == NORTH || car_direction == SOUTH) )
        {
            GzEvent.NoWait(traffic_light);
            MinorTimer.stop();
            MinorTimer = new SmallTimer(200,this);
        }
        /* car going at EAST or WEST direction and so is the current
           traffic light for the intersection. */
        else if ( traffic_light == EWDirection &&
                  (car_direction == EAST || car_direction == WEST) )
        {
            GzEvent.NoWait(traffic_light);
            MinorTimer.stop();
            MinorTimer = new SmallTimer(200,this);
        }
        /* car direction and traffic light direction does not match,
           car must wait until the light changed. */
        else {
            try {
                GzEvent.Stop(traffic_light);
                wait();
```

```
                GzEvent.Go(traffic_light);
            } catch (Exception e) {}
        }
    }
}
```

In the `enter()` method of the Intersection object, the first two '`if`' statement blocks handle the cases where the car direction matches the traffic light direction. In these two `if` statement blocks, the old `SmallTimer` is killed and a new `SmallTimer` is started as per discussion in section one, and execution continues without blocking. In the third `if` statement block, the execution of the `enter()` method is blocked on the call to the `wait()` method. This car will remain blocked until another thread, i.e., the timer thread, acts on the `traffic_light` variable and invokes the `notifyall()` method. (See Appendix A on the `wait()` and `notifyall()` mechanism for synchronizing access to shared variables from multiple threads.)

For the timer objects, we define the `LightChanged` event. This event occurs when the timer acts on the traffic light variable of the Intersection object. We also defined the `SmallTimer` and `LargeTimer` events to indicate when a timer is instantiated. The following code shows the locations where we place the `LightChanged` event in the `LargeTimer` object:

```
public class LargeTimer extends Thread {
    private int timeslice;
    private Intersection in;
    . . .
    public void run()
    {
       try
       {
           /* Sleeping until timeslice expired */
           sleep(timeslice);
           /* Enter critical section and switch the traffic light
              variable of the Intersection object.  Notify any other
              threads waiting on the Intersection object that
              traffic light has changed. */
           synchronized (in)
```

```
            {
                in.MinorTimer.stop();
                if (in.traffic_light==NSDirection) {
                    in.traffic_light = EWDirection;
                    GzEvent.LightChanged(EWDirection);
                    in.notifyAll();
                }
                else
                {
                    in.traffic_light = NSDirection ;
                    GzEvent.LightChanged(NSDirection);
                    in.notifyAll();
                }
                /*  Reset all timers prior to exiting. */
                in.MajorTimer = new LargeTimer(800,in);
                in.MinorTimer = new SmallTimer(200,in);
                GzEvent.closeThreadData();
            }
        } catch (InterruptedException e) { }
    }
}
```

Upon the expiration of the timeslice, the timer (i) toggles the traffic light variable of the Intersection object, (ii) notifies any other threads waiting for the traffic light to change, and (iii) resets the Large and Small timers controlling the Intersection object before exiting.

The event specification for the `StopLight` program is shown below. Additional attribute information is recorded for some of the events. For examples, for events associated with the car objects, the direction of car movement is recorded; for events associated with the intersection and timer objects, the direction of the traffic light is saved to the log files.

```
graze msgs
. . .
event Stop(traffic_light_direction) = symbol(plus);
event Go(traffic_light_direction) = symbol(diamond);
event NoWait(traffic_light_direction) = symbol(x);
event SmallTimer(traffic_light_direction) = symbol(x);
```

```
event LargeTimer(traffic_light_direction) = symbol(plus);
event LightChanged(traffic_light_direction) = symbol(diamond) ;
event ArriveIntersection(car_direction) = symbol(x) ;
event LeaveIntersection(car_direction) = symbol(plus) ;
event StartMotor = symbol(box);
. . .
end msgs.
```

From the event definitions, we specify the following intervals to be constructed and visualized:

```
graze msgs
. . .
interval StartMotorArrInt [StartMotor, ArriveIntersection]=rectangle;
interval ArriveLeave [ArriveIntersection,LeaveIntersection]=rectangle;
interval ArriveAndStop [ArriveIntersection, Stop]=rectangle;
interval StopAndGo [Stop, Go]=rectangle;
interval GoAndLeaveInt [ Go, LeaveIntersection ]  = rectangle;
interval STimerLtChanged [ SmallTimer, LightChanged ] =  rectangle;
interval LTimerLtChanged [ LargeTimer, LightChanged ] =  rectangle;
interval LtChangedAndGo [ LightChanged <- Go] Match = line;
interval NoWaitSmallTimer [NoWait,SmallTimer] Match = line;
. . .
end msgs.
```

`StartMotorArrInt` shows the interval between when a car starts and when it reaches the intersection. The `ArriveLeave` interval depicts cars that arrive at the intersection and go through it without stopping, whereas `ArriveAndStop` interval shows cars that arrive at the intersection and wait for the light to changed. The interval `StopAndGo` shows how long a car spends waiting for the light to change. The `STimerLtChanged` and `LTimerLtChanged` depict the intervals between when the timer is started and when it actually acts to change the traffic light. The `LtChangedAndGo` and `NoWaitSmallTimer` intervals associate events occurring on different threads.

3. Visualizing the Result

We used two different traffic patterns to test the working of Glimpse with the StopLight program. The traffic patterns are configured by setting the departure

time and the direction of the cars. In test case one, cars are initialized with random departure time and different directions. In test case two, a stream of cars traveling along the same direction is started at fairly close intervals. This steady stream of traffic flow is interspersed with a few cars traveling along the other direction. From the previous discussion on the StopLight program, we would expect that in test case one, where cars arrive randomly at the intersection while traveling at different direction, it is more likely that we should see the SmallTimer acts to switch the traffic light. In test case two, a steady flow of cars traveling along the same direction arriving at the intersection will tend to prevent the SmallTimer from changing the traffic light, but will favor the LargeTimer.

We ran the `StopLight` program with the two test cases multiple times on a SGI workstation (with the Java SDK 1.1.6 environment). The results vary slightly from run to run, but the general pattern is consistent between different runs. The following plots show the representative results.

a. Experiment One

In Figure 7, the interval `StartMotorArrInt` are plotted using the *gorge* visualization tool. The symbol `x` at the right edge of the horizontal bar indicates when a car arrives at the intersection. Because of the different departure times assigned to the cars, car objects instantiated later in the main program can reach the intersection earlier. For example, car 4 reaches the intersection before car 3.

In Figure 8, we added the `StopAndGo` interval to the plot. The `StopAndGo` interval depicts a car that must stop at the intersection for the light to change. From the plot, we see that car 1, 3, 5, 9, 10, 12 stopped at intersection before they were allowed to continue through. Next, we add the events associated with timer objects to the plot. The events `LargeTimer` and `SmallTimer` are depicted by the symbol `+` and the symbol `x` respectively. The first occurrence of a pair of `LargeTimer` and `SmallTimer` are associated with the instantiation of the `Intersection` object. These two events are shown at top of the plot (Figure 9) before the horizontal bars

Figure 7. Plot of `StartMotorArrInt` intervals for experiment one. The horizontal bar indicates the interval between when a car is started and when it reaches the intersection. The time scale increases to the right. There are 13 cars in this test case; they are displayed from top to bottom.

Figure 8. Plots of the `StartMotorArrInt` and `StopAndGo` intervals for experiment one. The StopAndGo intervals are bound by `+` and `diamond` symbols. Car 1, 3, 5, 9, 10, 12 have `StopAndGo` interval.

associated with car objects. Subsequently, we saw more instances of the `SmallTimer` event further down on the plot. The occurrences of these `SmallTimer` events are made clear in the next figure.

In Figure 10, we added the interval `NoWaitSmallTimer` to the plot. The `NoWaitSmallTimer` interval is represented by the vertical line that connects the right edge of the `StartMotorArrInt` interval in a car thread to the symbol `x` denoting the instantiation of a new `SmallTimer` object in a timer thread. There are 7 `NoWaitSmallTimer` intervals originating from car 2, 4, 6, 7, 8, 11, and 13; they correspond to cars that did not have to wait at the intersection for the light to change. Recall from the discussion of the *StopLight* program, cars that go through intersection without stopping are extending the duration of the traffic light in their direction by resetting the controlling timers. The interval `NoWaitSmallTimer` corresponds to this action, and the *gorge* provides a visual representation of this program behavior.

In Figure 11, We add `StimerLtChanged` and `LtChangedAndGo` intervals to the plot. The `StimerLtChanged` represents the interval between when a small timer is created and when its timeslice value has expired, at which time the small timer changes the traffic light direction. The `StimerLtChanged` interval is depicted as horizontal bars bounded by the `x` and `diamond` symbols. These `StimerLtChanged` intervals are shown below the car threads in Figure 11, except for the first instance of `StimerLtChanged` interval which is created by the `Intersection` object. From the plot we see that not every small timer thread contains a `StimerLtChanged` interval. This is because some small timers are preempted by cars going through the intersection, as discussed in the previous paragraph. The `LtChangedAndGo` represents the interval between when a car object stops at an intersection and when it starts moving again as the result of the expiring timer changing the traffic light direction. The `LtChangedAndGo` is depicted as vertical line connecting the right edge of the horizontal bars of the `LtChangedAndGo` intervals to the right edge of the `StopAndGo` intervals in the car threads. As can be seen from Figure 11, `LtChangedAndGo` intervals

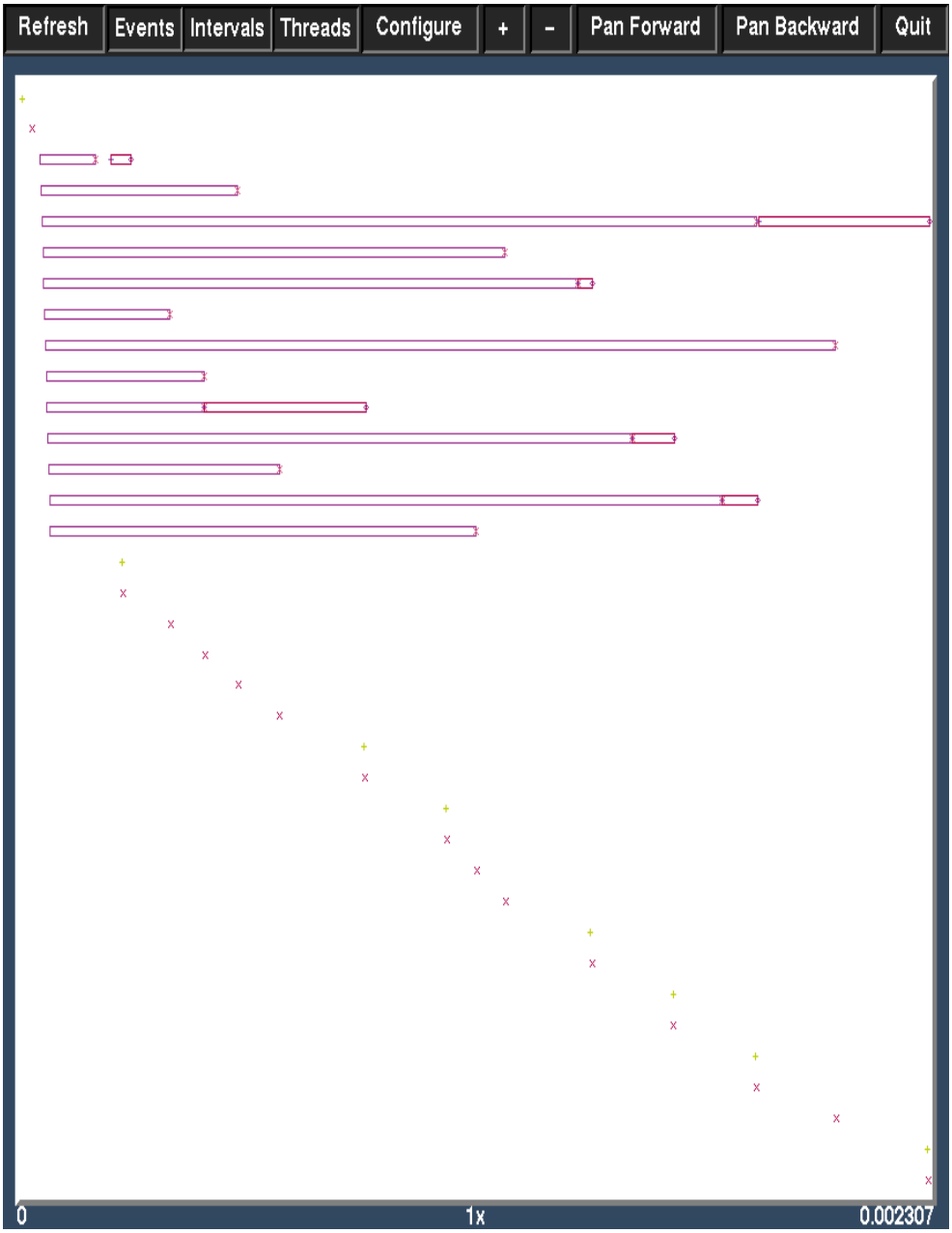


Figure 9. Plot of `LargeTimer` and `SmallTimer` events with `StartMotorArrInt` and `StopAndGo` intervals in experiment one. The symbol `+` and symbol `x` depict when the `LargeTimer` and `SmallTimer` objects are instantiated in the `StopLight` program. The first pair of timers is created (shown at the top of the plot) when the `Intersection` object is initialized, that is before any car objects are created.

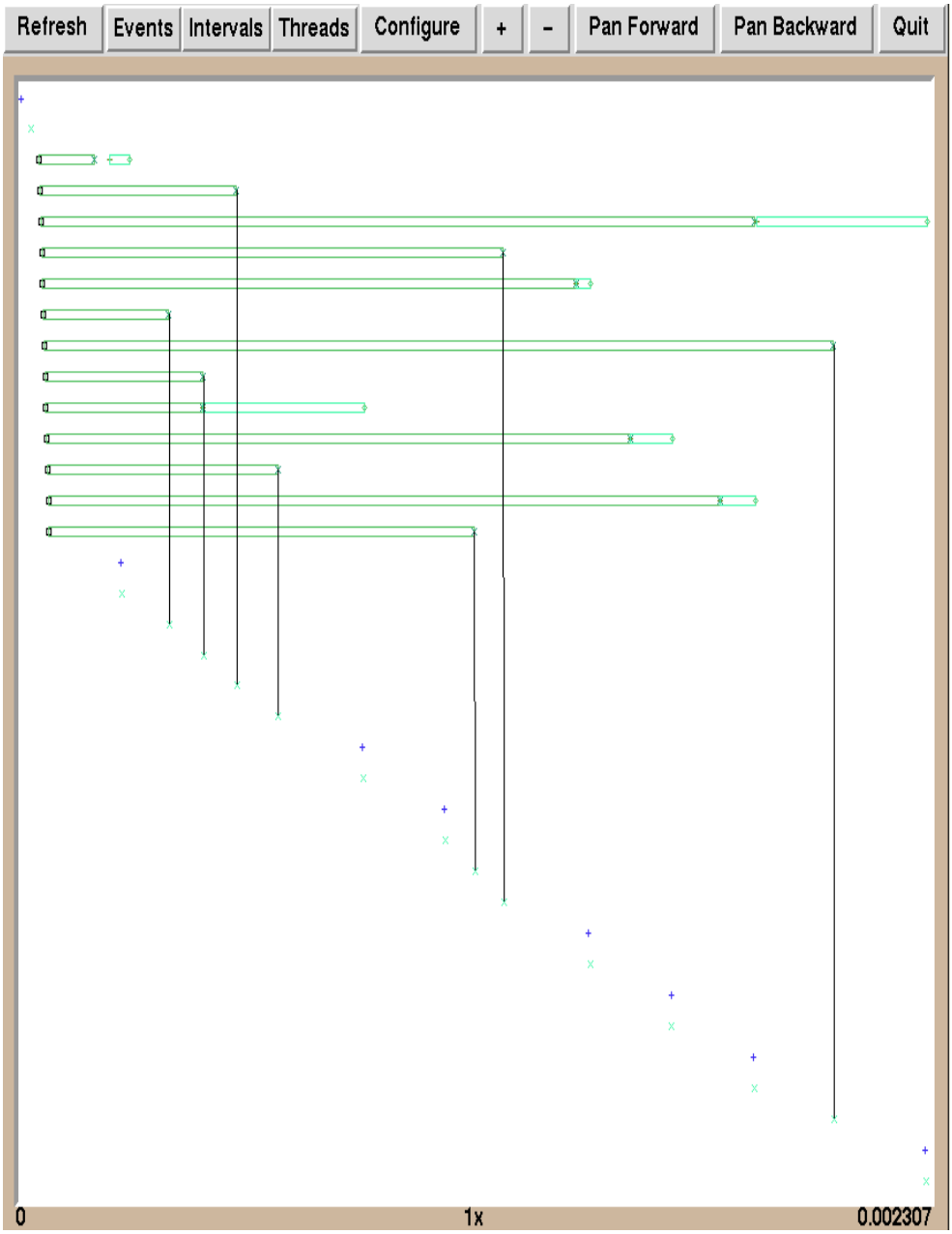


Figure 10. `NoWaitSmallTimer` interval is represented by the vertical line connection between the symbol `x` in the car thread and symbol `x` in the `SmallTimer`. There are 7 `NoWaitSmallTimer` intervals, originating from cars 2, 4, 6, 7, 8, 11, and 13. These intervals represent the new `SmallTimer` objects created by cars that did not have to stop at the intersection.

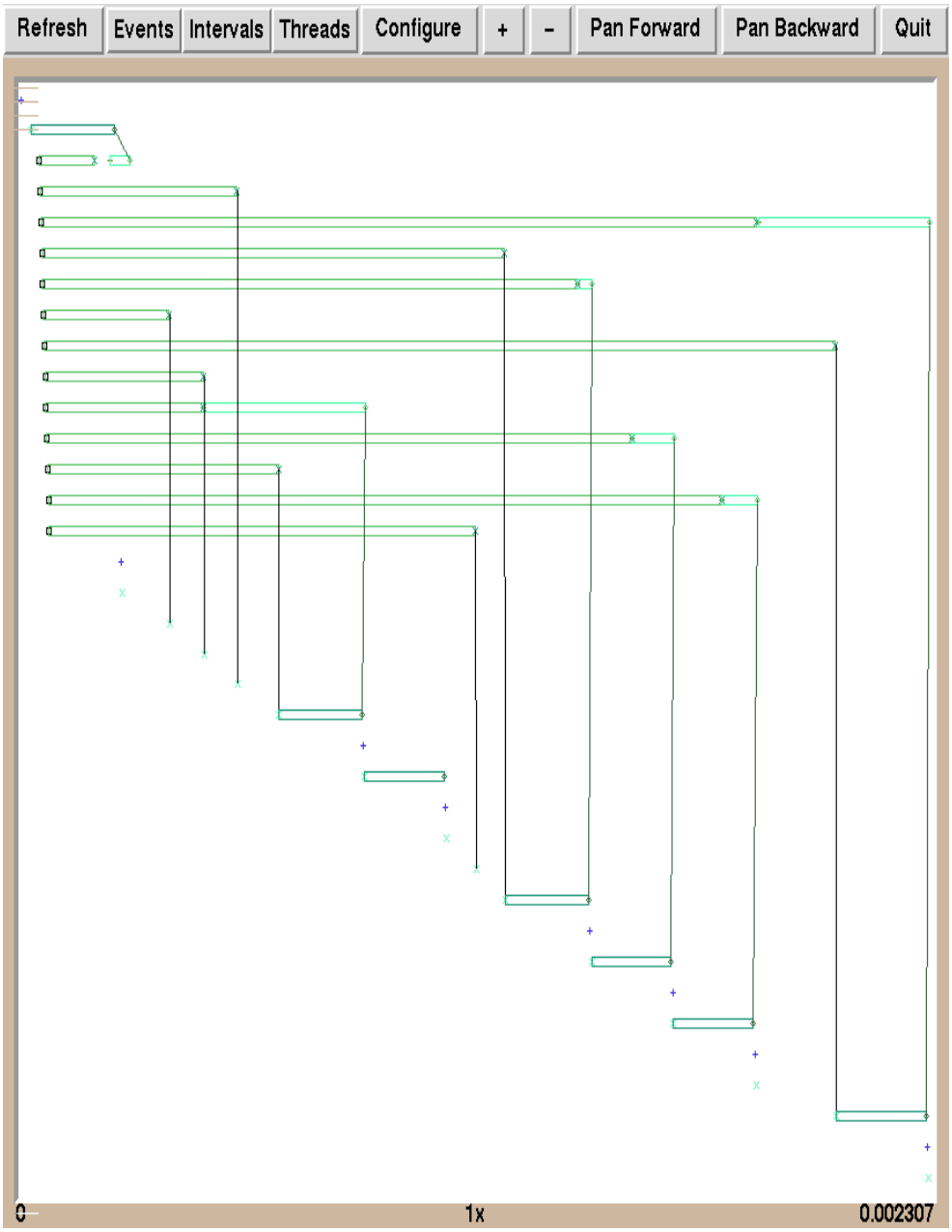


Figure 11. Interval `StimerLtChanged` is shown as the horizontal bar (between symbol `x` and symbol `diamond`) in timer threads. These intervals indicate the small timers that change the traffic light as the result of its timeslice value expiring. The interval `LtChangedAndGo` is shown as a vertical line from the right edge of the `StimerLtChanged` interval of the timer thread to the right edge of the `StopAndGo` interval of the car threads. These intervals represent the notification received by the car threads when the traffic light is changed by the timer threads.

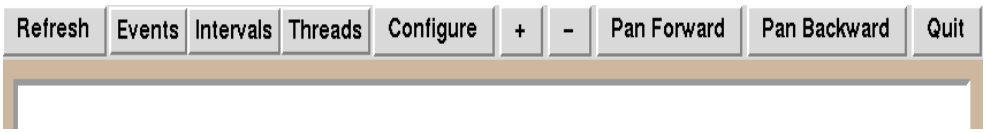
connects car 1, 3, 5, 9, 10, and 12 to `StimerLtChanged` interval. These are cars that were waiting at the intersection for the light to change. The interactions between the timer threads and car threads are consistent with the traffic distribution used in this test case and the design of the *StopLight* program. The visualization of intervals and events allow us to correlate the execution of the application program with the semantics of the program code.

b. Experiment Two

In this experiment, we execute the annotated test program using a different traffic pattern. This traffic pattern consists of cars arriving at the intersection at a constant interval, as shown by the `StartMotorArrInt` intervals in Figure 12. In Figure 13, we added the `StopAndGo` interval to the plot. We see that cars 1, 5, 12, and 13 must stop at the intersection before proceeding.

In the next Figure (Figure 13), the events associated with the creation of `SmallTimer` and `LargeTimer` are shown. These events are associated with the car that did not have to stop at intersection. The interval `NoWaitSmallTimer` in Figure 15 clearly shows the relationship of car threads creating the new timer threads.

The difference between experiment one and experiment two is that in experiment two only one of the `SmallTimer` threads ever reaches expiration. For the experiment two, there is a steady stream of cars going through the intersection, so the exiting `SmallTimer`, except for the last one, is always being re-started. Recall that from the discussion of the *StopLight* program, if cars repeatedly passing through the intersection constantly extend the traffic light, at some point the `LargeTimer` will act to change the traffic light. This is indeed what is shown by the `LtimerLtChanged` interval in Figure 16. The LargeTimer expired and allowed car 5, which has been waiting at the intersection, to proceed. When the `LargeTimer` expired, it created a pair of `SmallTimer` and `LargeTimer`. The event `SmallTimer` and `LargeTimer`, and the interval `StimerLtChanged` are shown at the bottom right of Figure 16. The expiration of the last `SmallTimer` allows cars 12 and 13 to proceed. This is shown by


Refresh
Events
Intervals
Threads
Configure
+
-
Pan Forward
Pan Backward
Quit